Tool-supported Refactoring of Aspect-oriented Programs

Jan Wloka
Rutgers University
Piscataway, USA
jwloka@cs.rutgers.edu

Robert Hirschfeld
Hasso-Plattner-Institut
Potsdam, Germany
hirschfeld@hpi.uni-potsdam.de

Joachim Hänsel
Fraunhofer FIRST
Berlin, Germany
joachim.haensel@first.fraunhofer.de

Abstract

Aspect-oriented programming languages provide new composition mechanisms for improving the modularity of crosscutting concerns. Implementations of such language support use advanced program representations, like abstract syntax trees or stack traces, to enable an indirect specification (pointcut) of executions of program elements at which aspect code (advice) is invoked. During the evolution of a program, this representations will change and, hence, advice may not be executed as intended by the developer.
In this paper we present a tool-supported refactoring approach that addresses this evolution problem by automating the detection of change effects on pointcuts and the generation of pointcut updates. A new model for decomposing pointcuts into simpler expressions is used as the base for deriving the change impact on pointcuts. Based on this model, we show how program analysis can detect affected or even broken pointcuts, how suitable pointcut adjustments can be derived, and when developer feedback is unavoidable.

Categories and Subject Descriptors D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—program analysis

General Terms Algorithms, Measurement, Languages

Keywords Software refactoring, Aspect-oriented programming, Change impact analysis, Static program analysis, Code generation

1. Introduction

Aspect-oriented programming (AOP) has been proposed for improving the modularity of crosscutting concerns. It introduces new adaptation mechanisms that allow for structural extension of implementation modules and adaptation of existing program behavior. The adaptation mechanisms are often supported by new language constructs, such as pointcut and advice. A *pointcut* specifies where and when an *advice* is executed by selecting well-defined points in the program execution, so called *joinpoints*. Every time a joinpoint is reached, some dedicated runtime support ascertains matching pointcuts. If a matching pointcut is defined, every bound advice is executed. With this complex but powerful mechanism, aspects can declare an adaptation without modifying the source of the adapted implementation module. To select a certain set of joinpoints, a pointcut can specify structural and behavioral properties of the program that have all desired joinpoints in common. These properties are made accessible by the implementation of the programming language, which provides access to static and dynamic program representations, such as the program's abstract syntax tree, the type hierarchy, or the stack trace. A pointcut expresses a joinpoint property, like the name of a method, or the declaring type of a method, and ties the advice execution closely to the representations used by the language implementation.

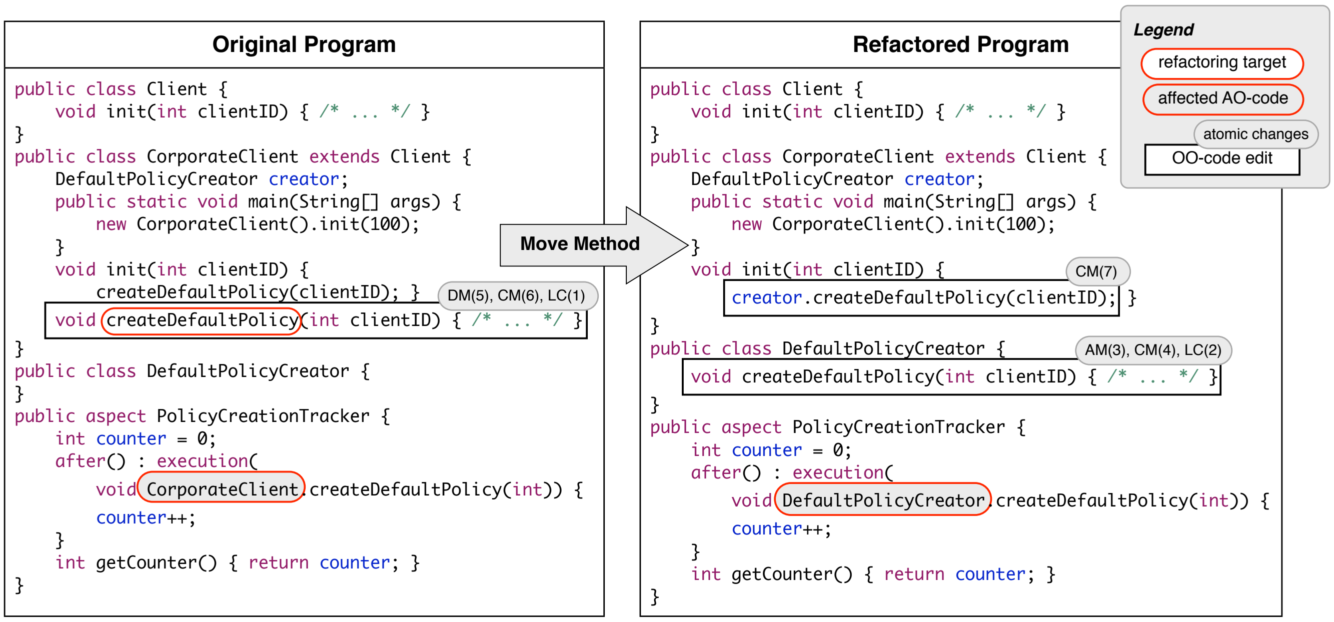
Like all software systems, aspect-oriented programs have to evolve over time. Developers have to adapt the program to changing requirements, fix bugs, or add new functionality. In today's constantly changing environments, the evolvability of a program is crucial. A widely adopted process for supporting evolution, mainly applied to object-oriented systems, is called software refactoring. The term is commonly used as the process for improving the design of existing code without altering its external behavior [15, 8]. Tool support for refactoring minimizes the effort and prevents the introduction of new errors in a manual application of the refactoring steps. A refactoring tool determines all change effects and estimates whether a change would alter the program behavior before a particular change is performed.

With the introduction of pointcuts even very local changes, such as renaming or inlining a local variable, can have global effects on a program behavior. Most modifications of program elements that are referenced by a pointcut can break the specification of a joinpoint property and cause unexpected advice execution.

In this paper, we present tool-supported refactoring that preserves the joinpoint properties specified in pointcuts, and provides automated adjustments of pointcuts in case a specified property is invalidated. Our approach allows for structural improvements of the design, but helps to preserve existing pointcuts. It guides the developer in deciding whether change effects on existing pointcuts invalidate them, and how pointcuts can be adjusted to restore the original program behavior. In particular, we present as the contributions of this paper:

- a meta-model for pointcut representations as a general basis for AOP tool-support,
- a change impact analysis for pointcuts based on this meta-model, and
- a heuristics-based impact assessment to automate the detection and adjustment of invalidated pointcuts.

The paper is structured as follows. In Section 2 we present a motivating example. In Section 3 we introduce our refactoring approach, describe our meta-model for representing pointcuts, explain the change impact analysis, and introduce an approach for automating the pointcut update decision. In Section 4 we present three experiments in which we evaluated our approach, discuss related work in Section 5, and conclude in Section6.



Original Program:

```
public class Client {
    void init(int clientID) { /* ... */ }
}
public class CorporateClient extends Client {
    DefaultPolicyCreator creator;
    public static void main(String[] args) {
        new CorporateClient().init(100);
    }
    void init(int clientID) {
        createDefaultPolicy(clientID); }
    void createDefaultPolicy(int clientID) { /* ... */ }
}
public class DefaultPolicyCreator {
}
public aspect PolicyCreationTracker {
    int counter = 0;
    after() : execution(
        void CorporateClient.createDefaultPolicy(int)) {
        counter++;
    }
    int getCounter() { return counter; }
}
```

Refactored Program:

```
public class Client {
    void init(int clientID) { /* ... */ }
}
public class CorporateClient extends Client {
    DefaultPolicyCreator creator;
    public static void main(String[] args) {
        new CorporateClient().init(100);
    }
    void init(int clientID) {
        creator.createDefaultPolicy(clientID); }
}
public class DefaultPolicyCreator {
    void createDefaultPolicy(int clientID) { /* ... */ }
}
public aspect PolicyCreationTracker {
    int counter = 0;
    after() : execution(
        void DefaultPolicyCreator.createDefaultPolicy(int)) {
        counter++;
    }
    int getCounter() { return counter; }
}
```

Figure 1. Example program, illustrating the change effects of a Move Method refactoring.

2. Motivating Example

In the following example, we illustrate the differences between refactoring object-oriented and aspect-oriented programs, as well as how the specification-like nature of pointcuts influence behavior preservation during a refactoring.

The example application in Figure 1 is implemented using AspectJ [28]. It can be seen as part of an insurance application that manages clients and contracted insurance policies. The program part consists of three classes, `Client`, `CorporateClient` and `DefaultPolicyCreator`, as well as one aspect `Policy-CreationTracker`. The class `Client` implements a method `init(int)`, which initializes the default policy for newly created client objects. The class `CorporateClient` extends class `Client` and redefines this method to invoke the local method `create-DefaultPolicy(int)`. The static method `main(String[])` invokes this redefinition with some data. This program is now restructured using the *Move Method* refactoring [8]. The refactoring is applied to move the method `createDefaultPolicy(int)` from class `CorporateClient` to class `DefaultPolicyCreator`. Figure 1 shows the particular changes caused by the refactoring in annotated boxes.

Standard refactoring tools, like the Eclipse JDT [7], use advanced program representations, such as an abstract syntax tree (AST) or the static type hierarchy, to check syntactic and semantic properties of the program for preserving its behavior. For object-oriented programs the properties relate to inheritance, scoping, type compatibility and semantic equivalence of references and methods, i.e., they are related to fundamental concepts of the programming language that is used to define the program behavior [15]. These properties must not be violated, if they exist in a program. Most refactoring tools implement explicit checks for these properties to determine behavioral changes. In this way, the preservation of the program behavior is automated by refactoring tools.

In the example, the refactoring tool checks, e.g., whether the target method `createDefaultPolicy(int)` is redefined in sub-classes, if it has incompatible control flow dependencies, or whether the class`DefaultPolicyCreation` already contains a method with the same signature, before it allows the developer to apply the refactoring. Also, references to the refactoring targets, such as method calls, are considered in the refactoring transformations. The resulting program in the example is then accomplished by copying the target method to class `DefaultPolicyCreation` and updating the method call in method `init(int)`.

This (object-oriented) refactoring does not consider any aspect composed with the program. The `PolicyCreationTracker` aspect in the example program simply counts the number of associated policies, by increasing a counter after a *policy-creation-method* was invoked. An aspect-aware version of this refactoring would consider the program behavior that is defined by aspects and in particular the advice invocations that are bound by pointcuts, as well[1]. New constraints need to be defined that preserve the advice invocations which lead to the composed behavior.

The aspect in the example defines the following pointcut:

```
pc1() : execution(
    void CorporateClient.createDefaultPolicy(int))
```

It defines that the advice is invoked directly after every execution of the method `CorporateClient.createDefault-Policy(int)`. Since the pointcut specifies a single method in a similar way as it is done by symbolic references, an aspect-aware version of the refactoring could easily determine that this pointcut has to be adjusted. A simple name lookup could identify the matching element for the specified method signature, and an adjustment would just replace the declaring type name. The updated pointcut as shown in Figure 1 would look very similar to the original specification and restore the original program behavior.

However, most pointcut languages provide developers with access to more joinpoint properties than just element names, and additionally allow for under and over-specification. A developer may specify a joinpoint property incompletely (under specification), so that executions of multiple elements in the program can be selected by a pointcut. For example, if the following pointcut is considered for the program in Figure 1:

[1] Structural extension mechanisms of aspects, such as inter-type declarations, have to be preserved by refactoring, too. Since this paper concentrates on a proper handling of pointcuts, this is considered to be out of the scope.

```
pc2() : execution(* Client+.*(..))
```

This pointcut would select all method executions of every method that is defined in class `Client` or its sub-classes. It does not state explicitly which methods it selects, rather than implicitly expect certain methods within the defined scope. A refactoring tool cannot simply adjust such a specification by replacing the signature pattern. A more sophisticated approach is required that involves the evaluation of the pointcut to determine whether matches of newly added or removed elements cause expected targets for an advice invocation. This also covers situations in which the composed behavior might intentionally be altered. The advice of the example is used in a logging scenario where executions of every method defined within class `Client` or its sub-classes are monitored. The refactoring moves the method `createDefaultPolicy(int))` to a different class and excludes it from the set of valid matches, which could be intended by the pointcut's developer.

In addition to under-specified properties, pointcuts can also refer to highly dynamic joinpoint properties. Consider the following pointcut, which selects only method calls occurring within a particular control flow at runtime:

```
pc3() : cflow(call(* Client+.init*(..)))
&&  execution(void createDefaultPolicy(int))
```

Such dynamic properties impose additional challenges to the analysis capabilities of refactoring tools. The tool has to create or maintain every representation that is used to compute such dynamic properties. Otherwise it would not be possible to detect a potential impact on the composed program behavior. Moreover, the use of dynamic properties complicates the computation of pointcut adjustments. Dynamic properties refer to a specific program behavior. If for some reasons this behavior was changed, the tool is asked to propose a pointcut that selects the new behavior. An automated generation of a specification that captures a particular behavior is, however, more difficult than capturing a particular structural property. In the example, the refactoring moves the `createDefaultPolicy(int)` method into a new containment scope. The pointcut uses this method to denote a certain control flow, which is not changed by the refactoring. Hence, the refactoring has no effect on the set of joinpoints selected by this pointcut.

The joinpoint properties specified in pointcuts are properties of the program that are expected by existing aspects. Refactorings for aspect-oriented programs have to preserve these properties, in a similar way as existing refactorings preserve program properties for object-oriented programs. A refactoring for Java, e.g., ensures that a method call invokes the same method implementation by protecting the involved type hierarchy and method redefinitions. Pointcuts of present AOP approaches use additional and more dynamic program properties that have to be considered when aspect-oriented programs are refactored.

To this end, a refactoring tool for aspect-oriented programs needs access to program representations that are additionally used by implementations of AOP languages, and it must be enabled to validate every joinpoint property specified in pointcuts. Furthermore, the tool cannot preserve the program behavior in all cases. A pointcut may capture only those program elements that exhibit a particular property, and exclude others by intention. If a refactoring alters this property, the pointcut should not match the modified elements in the refactored program. The resulting loss of captured joinpoints would cause an alteration of the composed program behavior. Such alterations can be intended, and thus have to be considered when refactoring aspect-oriented programs.

3. Aspect-aware Refactoring

Unlike approaches to aspect-oriented refactoring which use the aspect-oriented modularization concept for establishing new refactoring opportunities, we focus on a problem that is fundamental to refactoring in general, when it is applied to an aspect-oriented program: *aspect awareness*. Aspect-aware refactorings extend (object-oriented) refactorings [17], in order to make them aware of aspect bindings. The main goal of this approach differs from existing refactoring approaches regarding behavior preservation. In aspect-aware refactoring it is determined whether affected pointcuts allow for behavior alterations. If, for example, a pointcut intentionally under-specifies a joinpoint selection a refactoring may propose to accept altered matches, and thus an altered behavior. In cases where affected pointcuts are considered as invalidated, the proposed adjustment not only preserves the program behavior, it also needs to preserve the pointcut's characteristics, i.e., the way in which it specifies the joinpoint properties.

Since we focus on a tool-supported refactoring approach, we extend a standard process for automated refactoring: Input Validation – Change Preview – Transformation [7]. In particular, this process is extended by integrating a change impact analysis for pointcuts into the *Change Preview* which is used to:

- identify pointcuts that are affected by a refactoring;
- locate pointcut parts that are invalidated by a refactoring;
- determine whether to keep the pointcut or to preserve the behavior;
- generate a pointcut preserving adjustment if possible.

Also the *Transformation* step is extended to apply proposed and accepted pointcut adjustments.

In the following subsections, we introduce a new meta-model that explicitly maps program changes to pointcut references, and a specific change impact analysis that leverages the model to reveal affected parts of a pointcut. We argue why aspect-aware refactoring needs to be based on heuristics and present two heuristics that propose whether a pointcut has to be updated. Based on the meta-model and the heuristics we show how invalidated pointcuts can be updated. The complete refactoring approach is described in [27].

3.1 Modeling Pointcut References

Pointcuts specify properties of joinpoints which are manifested by the implementation of the programming language using different program representations. Most approaches make use of the program's name space, code containment, static typing and usages of declaring elements, such as types, methods and fields [3, 4, 19]. More advanced approaches, such as [2, 16, 24], also use properties of dynamic representations, like the stack trace (cflow), execution trace (trace match) and object heap (instance values). Every used program representation provides additional information and makes the selection of joinpoints more powerful. However, the same information has to be evaluated in the refactoring process, otherwise, it cannot be determined whether a certain modification affects a specified property.

In Figure 2 we give an overview of the models introduced by the following sections. The figure illustrates how they are used to compute pointcut references and how program changes caused by a refactoring are mapped to the pointcut expressions, so that affected expressions can be ascertained.

3.1.1 Pointcut Model

Our analysis approach uses a *pointcut model* for evaluating pointcuts within a particular program. This model is built for every pointcut defined in the program. It abstracts from the concrete syntax of the employed pointcut language and represents every single specification of a property through a separate pointcut expression. A *pointcut expression* (PCE) refers to a single joinpoint property, or to be more precise, to a property of an element of a program repre-

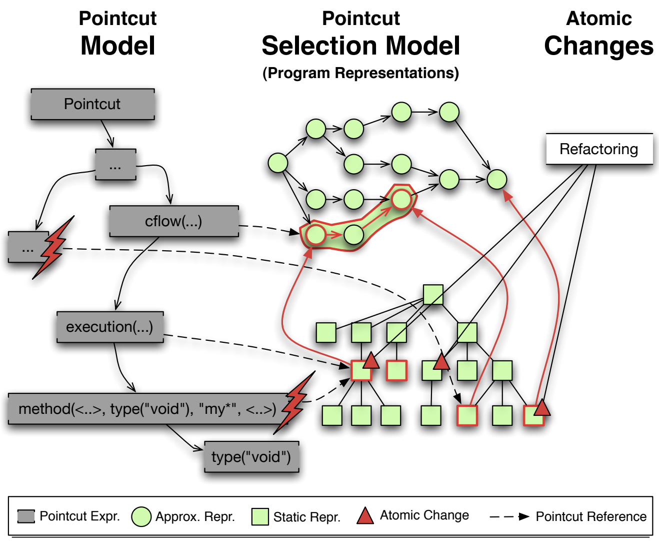


Figure 2. Overview of the pointcut meta-model

sentation that is used to identify the joinpoint property. The pointcut model is created by parsing the concrete syntax elements of a pointcut language and a subsequent decomposition of every partial specification into a tree of elementary pointcut expressions. The model represents pointcut expressions as nodes and the evaluation dependencies between them as directed edges.

Such an abstract, tree-based representation of expressions leads to three major advantages: (i) every expression refers to a single program representation, (ii) every expression holds the specification of a single property, and (iii) evaluation dependencies between the expressions are directly represented by the structure. This significantly simplifies the detection of program elements that correspond to a single property and allows for a distinct assessment of change effects on every single part of a pointcut. Pointcuts of every pointcut language, that specifies properties of joinpoints in a declarative way, can generally be translated in such a pointcut model. The model was evaluated for the pointcut language of AspectJ.

Notation Remarks. In the remainder of the paper we use a simple textual representation of the model to illustrate which properties are specified by pointcuts, how the specification is represented and which parts of the specification are affected by a change. An expression that specifies a single property has the form *expression(Property)* $\longrightarrow$ *Property*, whereas *Property* denotes the type of a joinpoint property, *<Property>* denotes an ordered list of properties, and *VARIABLE* represents a free variable parameter.

Static Properties. Static properties are evaluated on program representations that can be obtained from the program code, such as abstract syntax tree or static type hierarchy. The pointcut model supports the most common static properties like naming, code containment, and type relationships. For each property a separate expression is provided:

field(<Modifier>, TypeProperty, NAME) $\longrightarrow$ *FieldProperty*
get(FieldProperty) $\longrightarrow$ *Property*
call(MethodProperty) $\longrightarrow$ *Property*
within(Property, Property) $\longrightarrow$ *Property*
subtypes(TypeProperty) $\longrightarrow$ *TypeProperty*

The program's name space contains all named elements (declarations) of a program, such as packages, types, methods and fields. For each of these elements a separate expression is provided that takes the element signature as input and returns a name-based property. For example, the field expression above expects a list of modifiers, an expression that selects types, and a free variable *NAME*, which denotes the element's simple name and also allows wildcards for specifying partial names. The resulting property can be matched with field elements of a program's AST. In the same way, the pointcut model provides expressions for specifying usages of declared elements and containment-/hierarchy-based scopes. These expressions take a property as input and select a set of program elements, e.g., *get()* or *set()* select field accesses and *call()* a method call. Expressions like *within()*, *contains()*, or *subtypes()* specify the containment by a certain element or type hierarchy.

Dynamic Properties. Dynamic properties are not evaluated on the actual runtime structure, but on a conservatively approximated representation, i.e., the evaluation of expressions selects more elements than actually occur at runtime. The pointcut model supports dynamic properties related to dynamic typing and an execution's stack trace with expressions like:

this(TypeProperty) $\longrightarrow$ *Property*
target(TypeProperty) $\longrightarrow$ *Property*
cflow(Property, Property) $\longrightarrow$ *Property*

The dynamic type of the element currently under execution (*this()*), of the element targeted by the flow of control (*target()*) and the parameters (*args()*) is approximated using static inheritance relationships. These expressions return a set of possible types, rather than the actual dynamic type. The *cflow()* expression specifies containment in the stack trace using a call graph to approximate every possible stack trace. The expression receives two sets of elements and returns *true* if there exists at least one possible call path in the graph from an expression of the first set (*start-triggers*) to an expression of the second set (*end-triggers*). Also conditionals over runtime values can be used in pointcuts, which are approximated by returning *true*.

Composing Expressions. Pointcut expressions can be aggregated to specify more complex properties. In addition, logical combinations of *and* (||), *or* (&&), and *not* (!) can be used to filter elements from a given set or combine sets to a set union. With these compositors, almost every pointcut can be decomposed into a pointcut model. For example, the pointcuts from Section 2 can be represented as the following aggregations:
The model for pointcut `pc1()` consists of a *within()* expression (as root), a *type()* and a *method()* expression. Both sub-expressions specify the signatures completely, only the modifiers are left out.

within(type("CorporateClient"),
  method(<..>, type("void"), "createDefaultPolicy",
    <type("int")>))

The second pointcut `pc2()` just specifies that the methods in question have to be defined in a certain type hierarchy by defining the type's name:

*within(subtypes(type("Client")), method(<..>, *, *, <..>))*

The pointcut model for the third pointcut `pc3()` from the example section would look like the following:

cflow(
  call(
    within(
      subtypes(type("Client")),
      *method(<..>, *, "init*", <..>))),*
  method(<..>, type("void"), "createDefaultPolicy",
    <type("int")>))

3.1.2 Pointcut Selection Model

A pointcut resolver evaluates every pointcut model for a particular program and computes matching elements. Other approaches [12, 23] resolve only complete pointcuts and compute *joinpoint shadows* [12]. This static projection of a joinpoint into the program code is also computed by our resolver, but in addition it calculates matching elements for every partial aggregation of expressions following the inverse order of their evaluation dependencies. The resulting *pointcut selection* contains every element of a program that matches every partial aggregation of expressions. A program element that matches the root expression, i.e., all specified properties, is called *pointcut match*, whereas an element that matches a subset of expressions (partial aggregation), is called *property match*. Property matches are elements that are used to recognize a certain joinpoint property. They form the context in which the property can be identified. Changing such an element would make the same joinpoints unrecognizable if they occur. We call these elements *pointcut anchors* because the meaning of the pointcut is affected if they are changed. The pointcut selection model contains every element in the program that is selected by a pointcut, including elements selected by partial aggregations of sub-expressions.

3.1.3 Atomic Change Model

Program changes can vary in extent and complexity. They range from local edits to adaptations of multiple implementation modules. We have developed an abstract change representation that represents program edits by atomic changes. We use a similar notion for an atomic change as introduced by change impact analysis approaches for object-oriented programs [18, 21]. An *atomic change* represents program edits in terms of a program's AST. It covers modifications of program elements on every level of abstraction ranging from packages down to statements.

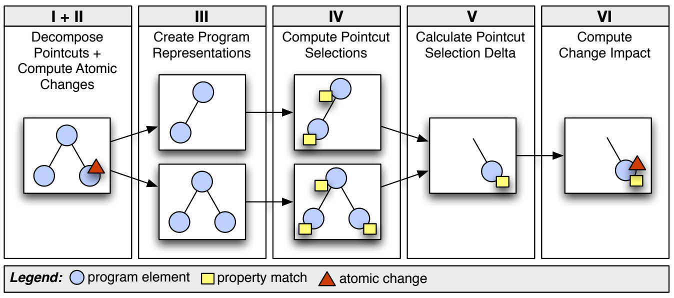
Since our impact analysis approach compares original and refactored program versions, only changes that affect the existence of the elements in an AST are of interest. Therefore, only changes that cause the creation or deletion of elements are considered, such as added type (AT), deleted type (DT), added method (AM), deleted method (DM), added statement (AS) and deleted statement (DS). Other edits, like rename or move, can be represented by these atomic changes if the analysis is aware of the transformation that causes an atomic change.

Considering the example program, the move of method `createDefaultPolicy(int)` illustrated in Figure 1 is represented by $AM(3)$ (the method with the new name) and $DM(5)$ (the method with the old name). Since the method is not empty and not private additional changes indicate method body modifications ($CM(4)$, $CM(6)$) and alterations in the lookup table ($LC(1)$, $LC(2)$).

Refactoring tools realize the individual refactoring steps by program transformations. These transformations modify a program in terms of the underlying AST and thus create, remove, move, or in case of declaration elements also rename, program elements. In our atomic change model, these kinds of transformations represent the reasons for an atomic change and are stored in the model if their changes affect a pointcut expression.

3.2 Change Impact Analysis for Pointcuts

The impact analysis presented in this section is able to detect whether a pointcut is affected by a refactoring. It can reveal the expressions of a pointcut that are affected by a refactoring and also determine which kind of change causes the effect. The primary goal of the analysis approach is to assess and classify change effects on pointcuts in aspect-oriented programs, as well as to compute adjustments for invalidated pointcuts, making effects on the program behavior undone. The analysis uses the models introduced in the previous section. We refer to the example program to illustrate the resulting information of each step.



I + II
Decompose Pointcuts + Compute Atomic Changes
III
Create Program Representations
IV
Compute Pointcut Selections
V
Calculate Pointcut Selection Delta
VI
Compute Change Impact
Legend:
program element
property match
atomic change

Figure 3. Overview of the impact analysis process.

3.2.1 Overall Analysis Process

The analysis process comprises *six*, sequentially performed, analysis steps. An overview of this process is given in Figure 3, illustrating the information gained from every analysis step.

(I) Decomposition of Pointcuts. In the first analysis step, we construct the *pointcut model*. All existing pointcuts are decomposed into elementary pointcut expressions. For instance, the pointcuts of the example are decomposed into representations as shown in Section 3.1.1. Then the evaluation dependencies between the expressions of a pointcut are determined, and every partial aggregation of them is computed. The resulting pointcut model represents a pointcut as a tree of pointcut expressions, using nodes to represent expressions (a leaf node indicates an independent expression) and directed edges to represent evaluation dependencies. Continuing with the example, the resulting model for pointcut `pc1()` consists of the following expressions along their evaluation dependencies[2]:

{*type("void"); type("int"); type("CorporateClient");*
method(<..>, type("void"), "createDefaultPolicy",
<type("int")>);
within(type("CorporateClient"), method(...))}

(II) Computation of Atomic Changes. Our analysis computes the atomic change model for the chosen refactoring by decomposing the modifications in atomic changes and by adding the responsible transformation kind as reason to the changes. The atomic change model is computed after the refactoring tool has performed the transformations virtually to produce the refactored program version. In Figure 1 the edit caused by the refactoring is illustrated with annotated boxes. Every box shows the atomic changes that are computed from the edit. The refactoring results in set {*AM(3), CM(4), LC(2)*} for the added method in class `DefaultPolicyCreator`, a *CM(7)* for the changed call in method`init(int)`, and set {*DM(5), CM(6), LC(1)*} for removing the original method.

(III) Creation of Advanced Program Representations. This step creates statically available representations for specified joinpoint properties using the *original* and the *refactored program* version. In particular, an abstract syntax graph (ASG) is computed, representing containment, inheritance and usage relationships, and partial call graphs (CGs) are created, representing call dependencies, of the program. The analysis creates only the program representations that are required for evaluating specified joinpoint properties, i.e., the kinds of pointcut expressions contained in the pointcut model.

(IV) Computation of Pointcut Selections. Our impact analysis uses the advanced program representations to evaluate the

[2] The expression *method(...)* is used to abbreviate the actual expression due to space limitations.

pointcut expressions. This *pointcut resolution* step evaluates every aggregation of pointcut expressions following the evaluation dependencies defined by the pointcut model. A pointcut resolver computes the program elements that matches the specified properties, or to be more precise, the nodes of the employed program structure which represent the program elements (pointcut selection). The pointcut `pc1()` matches executions of method `createDefaultPolicy(int)`, and additionally refers to this method, to class `CorporateClient`, and to the special types `int` and `void` as pointcut anchors.

(V) Determination of the Pointcut Selection Delta. This analysis step compares the pointcut selection models for the *original* and the *refactored program* and produces the *pointcut selection delta*. The delta contains all new and lost matches for the refactored program version, and thus represents the direct impact of a refactoring on pointcuts in the program. This delta may contain spurious effects (in both program versions) of the same program transformation. The actual delta is determined by locating every program transformation that causes new or lost matches of a pointcut expression. Since the transformation is explicitly available, for every lost match in the refactored program the corresponding added match in the same version can be located. If two corresponding matches can be found, both are removed from the pointcut selection delta. As result, only really new and lost matches remain in the impact representation for every pointcut. The move refactoring in Figure 1 causes altered matches of the following expression:

within(
type("CorporateClient"),
method(<..>, type("void"), "createDefaultPolicy",
<type("int")>))

(VI) Computation of Change Impact. The resulting change impact representation contains all information necessary to assess the extent of an impact. Three different kinds of information are used for this assessment: (i) the change reason, (ii) the specification quality of affected pointcut expressions, and (iii) their relevance to the pointcut. These impact measures are described in the following sections.

3.2.2 Impact Classification

Our change impact analysis computes an explicit representation of the impact, stating which kind of transformation causes new or lost matches for which expression of the pointcut. It can be used to assess the effects on the program behavior that is selected by a pointcut. Since pointcuts are declarative specifications, we can distinguish changes that (i) alter the program behavior that corresponds to the specification, and (ii) modify properties of program structures that are used to recognize this behavior. Both kinds of changes cause a different set of selected joinpoints and, thus, affect the composed program behavior. The former alters the behavior of the base program (rarely achieved through refactoring), which changes how often a specified behavior occurs at runtime. The latter changes properties that are specified by a pointcut in order to recognize the specified behavior. The same behavior cannot be identified by the pointcuts, because they expect joinpoints with the original, unchanged, properties.

Our analysis approach does not consider the first kind of changes, because all pointcuts still select the joinpoints in the execution of the base program that are associated with the same properties, even if they occur more or less often at runtime. For the second kind of changes we developed an impact classification, that categorizes the impact on affected properties in terms of their specification.

The most difficult part in measuring the impact of a change on a pointcut expression is to assess how much information the expression has to specify that it selects the changed element only. Another important issue is the approximation of dynamic properties by static representations. The *analyzability of properties*, however, cannot be measured directly. We only distinguish three kinds of dynamic properties: runtime value-based, dynamic type-based, and control flow based properties. Runtime values are considered as not analyzable, whereas dynamic types can be properly approximated by the corresponding static type hierarchy. Control flow properties are approximated with the call graphs as described above.

Table 1. Definition of the execution semantics measure.

Classifier	Kind of scope	Focus	Measure
behavioral scope	control flow	statement	100
	inheritance	method	
	containment	operation	
lexical scope		type	25
		package	5
unscoped	no	no	0

Specification Completeness. A pointcut can under-specify properties by using partially defined signature patterns [28]. The specification completeness indicates how complete signature patterns are defined. A name property is completely specified (100%) if any part of used signature patterns is fully defined. Incomplete parts, such as partial names or partial parameter lists, are counted with 50%, while undefined parts (wildcards) are considered with 0%. For aggregated expressions the completeness is measured by the average value of all sub-expressions.

Match Scope. An indicator for the importance of a matching element is the *match scope*. It indicates whether an affected pointcut expression is used to select a single or multiple elements. The match scope measures how many elements along the path that leads to a matching element within the lexical structure of a program are specified by the pointcut. For example, the call to method `init(init)` is captured by the *call()* expression of `pc3()`. The match scope indicates whether this expression specifies the call's method, its enclosing feature, the type declaring the feature and the type's package. The match scope measures how many parts along the scoping path are specified by a pointcut expression. The basic assumption behind this metric is the more a matching element is scoped by an expression, the more important is the match to the pointcut.

Execution Semantics. A pointcut can specify properties with a different degree of behavioral meaning. There are properties with no behavioral meaning, like an unscoped name, and properties with a specific meaning in the execution of a program, such as a kind of certain statement. We introduce a naive distance measure that indicates how close selected elements are related to a specific program behavior. Table 1 shows the particular metric values. The assumption behind this metric is that an altered match of a behavioral property (close to execution semantics) is more likely to accept as of a structural property, because modifications in the structure should not alter the behavior. Therefore, matches restricted by a *behavioral* scope are counted as 100% and unscoped matches as 0% of execution semantics. The two values in between are low indicators just to distinguish elements somehow related to a behavior from completely unrelated ones.

Degree of Dependency. The *nesting level* of a pointcut expression within a pointcut indicates the degree of its evaluation dependencies. Since the expressions in the pointcut model are nested like in an AST, an expression is more important to the evaluation result the deeper it is nested in the tree. A comparison of the nesting in pointcuts of multiple AOP projects has given a strong indication, even without hard evidence, that expressions with a deeper nesting than level 2 can hardly be differentiated in their importance for the pointcut's evaluation result [27]. We therefore distinguish the first three nesting levels in their importance with 0%, 50%, and 100%.

3.3 Update Decision Making

Any affected pointcut expression causes altered matches in the pointcut selection delta. Based on the four impact measures we define two heuristics that provide different indicators for whether an altered match invalidates a pointcut.

Specification Quality. A program element can be explicitly selected by a pointcut or just be one of numerous matching elements. The *specification completeness* metric is used measure how complete matching properties are specified. In addition, the *match scope* determines how much an expression restricts its selection to the element(s) matching in the program. Our impact representation associates a matching element (*pm*) with all corresponding expressions of a pointcut. The expression that specifies the maximum number of properties for this element is called ex_{max}. If a matching element is modified, the smallest sub-expression of ex_{max} that matches the element is the (smallest) directly affected part of the pointcut, called ex_{aff}. Assuming that fully scoped and completely specified matches are more desired by the developer than less precisely specified matching elements, we can define the specification quality SQ as follows:

$$SQ(pm) = \frac{SC(ex_{aff}) \cdot \frac{1}{2}(100 + MS(ex_{max}))}{100}$$

with $ex_{aff},\ ex_{max} \in PCE;\ pm \in PSM$

In the relation of specification completeness and match scope, is the specification completeness considered to be twice as important as the match scope, i.e., in the worst case a completely unscoped expression can reduce the value of the specification completeness by 50%. The specification quality is computed for the every altered match using the directly affected expression to measure how precise the changed element is specified, and the largest matching expression to measure how well it is scoped.

If the `pc2()` from the example is considered, the refactoring affects expression *within()*, which leads to *SQ(*`createDefaultPolicy(int)`*) = (SC(within()) * 0.5 (100 + MS(within()))) / 100 = (50 * 0.5(100 + 75)) / 100 = 44%.*

Expression Relevance. An affected pointcut expression can be more or less relevant for the evaluation of the complete pointcut. This heuristic assesses an expression's relevance using the *degree of dependency* and *execution semantics*. Based on these measures the expression relevance is defined as the zero-bounded difference:

$$ER(ex) = \begin{cases} DD(ex) - ES(ex) & \text{if } DD(ex) - ES(ex) > 0 \\ 0 & \text{if } DD(ex) - ES(ex) < 0 \end{cases}$$

with $ex \in PCE$

The assumption behind this heuristic is that an altered joinpoint match, representing a certain behavior, is more likely to be wanted if a behavioral property was changed rather than a structural (like naming). However, the deeper an expression is nested in the pointcut the more unlikely it is that effects on the resulting pointcut selection can be intended. For pointcut `pc2` the *ER* is computed for expression *within()* which results in *ES(within() - DD(within()) = 100 - 50 = 50%.*

With these heuristics we can automate the update decision making. Our approach follows two general assumptions: (i) elements that match precisely defined expressions (SQ = high) are wanted by the developer, and (ii) selections of deeply nested expressions must be preserved. For both heuristics we have defined an initial range that states how precise or relevant affected expressions have to be, so that they are considered as invalidated. Using these initial benchmarks we decide whether a pointcut PCE needs to be updated as follows:

$$\begin{aligned} &\textit{SQ(m) = 0\%} \rightarrow \begin{cases} \textit{NOUPDATE(ex)} & \text{in any case} \end{cases} \\ &\textit{0 < SQ(m) < 100\%} \rightarrow \begin{cases} \textit{NOUPDATE(ex)} & \text{if } \textit{ER(ex) < 60\%} \\ \textit{UPDATE(ex)} & \text{if } \textit{ER(ex) >= 60\%} \end{cases} \\ &\textit{SQ(m) = 100\%} \rightarrow \begin{cases} \textit{NOUPDATE(ex)} & \text{if added match} \\ \textit{UPDATE(ex)} & \text{if lost match} \end{cases} \end{aligned}$$

with $m \in PSM,\ ex \in PCE$

Using this table, our refactoring tool recommends to accept additional matches of unspecified or precisely specified expressions, and to accept lost matches of unspecified expressions. Moreover, altered matches of expressions with an average precision are only accepted if the expression has a relevance of less than 60%. In case of `pc2()` the tool would propose not to update, because of *ER(within()) = 50%.*

3.4 Generation of Updates

If the analysis has proposed to update an affected pointcut expression, the most straight forward approach is to exclude unwanted or include lost matches. A refactoring tool can add an extra pointcut expression at the same nesting level of the affected expression to the pointcut. The additional expression specifies a direct and fully specified exclusion or inclusion of altered matches. This direct exclusion/inclusion of individual matches is already proposed by other refactoring approaches for AOP [20, 10]. We can summarize this approach by two update patterns and define them as follows:

$pce(ap) = pce(ap)\ ||\ pce(ip) : ap \in PCE_{aff},\ ip \in PCE_{incl}$
$pce(ap) = pce(ap)\ \&\&\ !pce(xp) : ap \in PCE_{aff},\ xp \in PCE_{excl}$

This update approach, however, leads to several disadvantages, such as pointcut bloating. Every update adds new pointcut expressions to the existing pointcut, which makes the pointcut unrecognizable to its developer, already after a few updates.

A more sophisticated approach would check whether the affected pointcut expression can be completely replaced by a new expression. Such a replacement is possible in cases where all matches of the affected expression are altered and the replaced expression results in equivalent matches. It can be defined as:

$pce(ap) = pce(rp) : ap \in PCE_{aff},\ rp \in PCE_{repl}$

Our approach always tries to replace the affected pointcut expression. It detects the expression that is directly affected by a refactoring's transformation and determines if the refactoring splits the set of matching elements. If not, we can directly replace the affected pointcut expression with the same specification quality. The possibility to directly replace affected pointcut expressions is a prerequisite to preserve the pointcut within a refactoring process.

4. Evaluation

The refactoring approach developed in this work was evaluated using our refactoring tool SOOTHSAYER. In this section we introduce the tool, describe the employed evaluation methodology, explain expected results, and present three experiments in which independently developed AspectJ programs are refactored using our approach. We also discuss the evaluation results, expose the kinds of pointcuts that cannot be properly handled by our tool and elucidate reasons.

4.1 Refactoring Tooling

Our refactoring tool SOOTHSAYER supports our approach for refactoring AspectJ programs. It implements the presented analysis approach as an Eclipse plugin[3]. The plugin extends the Java refactoring capability of the Eclipse JDT, it provides an optimized structural representation of the source code and a call graph for approximating stack trace based pointcut expressions. SOOTHSAYER statically evaluates pointcut matches for most of the AspectJ pointcut designators, including dynamic designators, such as `cflow`, `this`, `target` and `args`. It creates the atomic change model for several Java refactorings, such as rename, move, extract and inline, determines the pointcut selection delta for the refactored program and computes the presented change impact representation.

SOOTHSAYER implements the presented pointcut update decision table and proposes pointcut updates for broken pointcuts. Considering the defined change classification, any affected pointcut expression is either not adjusted, broadened (to include lost matches), narrowed (to exclude new matches) or labeled as broken or unresolvable.

4.2 Methodology

We evaluated our approach using three different AspectJ applications. The goal of this evaluation was to validate the computation of the change impact, the update decision proposals, and the generated pointcut adjustments. More precisely, we wanted to show that our impact analysis approach is able to represent change effects on pointcuts concretely, so that a minimal-invasive update proposal can be inferred automatically. A pointcut can only remain recognizable to its original developer if its updates change the least possible number of expressions, i.e., are minimal-invasive. Another goal of this evaluation was to show that the smallest affected pointcut expression can be detected for a variety of refactorings. In particular, we are interested in understanding when a minimal-invasive update cannot be proposed, and for which kinds of pointcuts it is impossible.

The AspectJ applications were selected according to the characteristics of their pointcuts, rather than the application's size. The size may affect the scalability of our analysis approach, but this was not the focus of this evaluation. The major goal was to apply our approach to very different kinds of pointcuts. We have selected the refactorings and their targets in the source code in a way that at least some effect on one of the pointcuts in a program can be expected. Particularly, the target pointcuts have been chosen for whether a refactoring can affect:

- pointcuts that specify different properties of various program representations;
- pointcuts that specify these properties in different ways;
- pointcut expressions at different nesting levels of a pointcut.

Every program is tested by a suite of unit tests that reveals altered executions of existing advice. This way, unexpected side effects of a refactoring can be made visible. The test suite is run before and after a refactoring. The comparison of test failures demonstrates whether all expected change effects are detected, and if proposed pointcut adjustments restore the original behavior. The particular update decision is validated during a refactoring. Our impact analysis shows the effects on every pointcut and we have decided, whether altered property matches can be accepted or the pointcut has to be updated. In any case where an adjustment was proposed, the updated pointcut was tested with the refactored program. For cases where altered matches where supported to be accepted we run the test suite against the original pointcut in the refactored program.

The measured indicators and proposed updated decisions for all experiments are shown in Table 2. In the table as well as in the descriptions we use the following abbreviations: Specification Completeness (SC), Match Scope (MS), Specification Quality (SQ), Execution Semantics (ES), Degree of Dependency (DD), Expression Relevance (ER) and Match Impact (MI).

```
aspect Timing
  (Connection c) :
    target(c) && call(void Connection.complete())
  endTiming(Connection c) :
    target(c) && call(void Connection.drop())

aspect Billing
  (Customer cust) :
    args(cust, ..) && call(Connection+.new(..))
```

Figure 4. Pointcuts in Telecom application.

4.3 Experiment 1: Telecom Application

The *Telecom* application is a small AspectJ program that simulates telephone connections to which timing and billing features are added using aspects. It is available from AspectJ's website [3].

The `Timing` aspect manages the total time per customer and is added as a timer to each connection. It defines two pointcuts that intercept all executions of any call to two specific methods.

The `Billing` aspect calculates a charge per connection and builds upon the `Timing` aspect. It defines one pointcut that is used to receive the caller who pays for the call. The pointcut intercepts all invocations of any constructor in `Connection` (or its subclasses) and receives the constructor's first argument in a parameter.

(S1) Rename Method: "Connection.complete()". The *Rename Method* refactoring changes the name of method `complete()` to `completeConnectionCall()`. Our impact analysis indicates the loss of all matches of the completely scoped expression (MS=100%):

within(type("Connection"),
method(<..>, type("void"), "complete", <>))

The analysis further recognizes the pointcut expression *method(<..>, type("void"), "complete", <>)* as the smallest expression that is directly affected by the refactoring's *RENAME* transformation. This expression specifies the method signature completely, which results in SQ=100%. The analysis decides to "update the pointcut", and proposes to replace the affected expressions, since all of its matches are lost:

```
(Connection c): target(c)
&& call(void Connection.completeConnectionCall())
```

(S2) Rename Type: "Connection". The *Rename Type* refactoring changes the name of class `Connection` to `TelConnection`. Our impact analysis reveals lost matches of the expression *type("Connection")* in every pointcut of aspects `Timing` and `Billing` (cf. Figure 4). The lost matches are completely scoped (MS=100% through import statements) and the affected expressions are completely defined (SQ=100%). As result, a precisely defined match is lost, thus the analysis propagates to "update the pointcuts". The update computation proposes to replace the affected expression in each pointcut directly.

(S3) Inline Method: "Connection.complete()". *Inline Method* targets the method `complete()`, replacing every call to this method with its body, and removing the original method declaration. Our impact analysis detects lost matches of expression:

within(type("Connection"),
method(<..>, type("void"), "complete", <>))

[3] SOOTHSAYER is a research prototype. The current version is available upon request from the authors.

Table 2. Overview of all refactoring scenarios in the evaluation.

Scenario	SC	MS	SQ	DD	ES	ER	Matches	Transform.	Decision	Update
S1	100	100	100	100	0	100	lost	rename	update	replace
S2	100	100	100	100	5	95	lost	rename	update	replace
S3	100	100	100	100	0	100	lost	remove	update	cancel
S4	0	50	0	100	0	100	new	create	noupdate	-
S5	68	0	34	33	100	0	new	create	noupdate	-
S6	-	-	-	-	-	-	-	-	-	-
S7	100	100	100	67	100	0	-	move	-	-
S8	100	100	100	100	0	100	lost	rename	update	replace
S9	17	100	17	100	25	75	new	rename	update	exclude
S10	100	60	80	67	100	0	new	create	noupdate	-

Legend	
SC	Specification Completeness
MS	Match Scope
SQ	Specification Quality
DD	Degree of Dependency
ES	Execution Semantics
ER	Expression Relevance

It also reveals the expression *method(<..>, type("void"), "complete", <>)* as directly affected by the refactoring's *REMOVE* transformation. Our analysis propagates "update the pointcut" for preserving the original behavior. Since matches of removed elements cannot be recovered, the update computation proposes "cancel the refactoring".

4.4 Experiment 2: Spacewar Application

In the second experiment, we refactor an AspectJ implementation of the video game *Spacewar*. The program is slightly bigger than the *Telecom* application, but uses significantly more aspects for various functionalities in the application. It is also available from the AspectJ's website [3]. Most of the aspects use simple pointcuts similar to the pointcuts in the *Telecom* application, but the pointcuts of the aspects `Debug`, `DisplayAspect` and `Ship`, could lead to new refactoring situations (cf. Figure 5).

```
aspect Debug
allMethodsCut() :
execution(* (spacewar.* && !(Debug+ || InfoWin+))
    .*(..))

aspect DisplayAspect
(Display display) : call(void setSize(..)) &&
    target(display)
() : call(Display+.new(..))

aspect Ship
helmCommandsCut(Ship ship) :
target(ship) &&
  (call(void rotate(int))
     || call(void thrust(boolean)) || call(void
        fire()))
```

Figure 5. Pointcuts in Spacewar application.

The `Debug` aspect specifies debugging information that is displayed in the information window. Several of its pointcuts use nested, partially specified name patterns to select executions of different program elements in classes of package `spacewar` except if they are defined in the classes `Debug`, `InfoWin`, or its sub classes.

The aspect `DisplayAspect` defines two particularly interesting pointcuts. The first pointcut uses incompletely specified signature patterns to select all executions of `setSize()` method calls within classes of class `Display`, and the second pointcut intercepts all executions of any constructor call of class `Display` (or its subclasses).

The aspect `Ship` defines a pointcut that enumerates three different methods for intercepting their invocations. It restricts the scope in which methods with the specified signatures are selected through a dynamic type.

(S4) Extract Method from "Game.run()". The *Extract Method* refactoring is performed within method `Game.run()` to extract the statements at lines 84-90 into a new method (cf. [22]). It creates a new method named `createRobots()`, copies the selected statements into this method, and replaces the original statements with a method call to the newly created method. The impact analysis reveals a new match with MS=50% of expression:

within(package("spacewar"),
method(<..>, type(""), "*", <..>))*

of pointcut `allMethodsCut` in aspect `Debug`. It further detects that the sub-expression *method(<..>, type("*"), "*", <..>)* is directly affected by the refactoring's *CREATE* transformation. Our tool proposes not to update the pointcut, i.e., to accept the new property match, because this affected sub-expression is considered to be a wildcard (SQ=0%).

(S5) Extract Method from "Display.initializeOffImage()". The *Extract Method* refactoring is performed within method `Display.initializeOffImage()` to extract the image size setup (lines 73-76) into the new method `setSize(double, double)`. Our impact analysis reveals a new match of the unscoped expression (MS=0%):

call(within(subtypes(type("Display")),
method(<..>, type("void"),"setSize",<..>)))

of the first pointcut defined in `DisplayAspect` (cf. Figure 5). It detects that this expression is directly affected by the refactoring's *CREATE* transformation, and computes a SQ=34%. Since SQ is neither 0% nor 100% our analysis also considers the relevance of this expression ER=0%. Our tool proposes not to update the pointcut.

(S6) Move Type "Display". The refactoring moves the class `Display` from package `spacewar` to package `spacewar.core`. This refactoring basically changes import declarations and fully qualified class names. Our impact analysis recognizes this change by comparing of the pre- and post-refactoring versions but it does not detect any effect on existing pointcuts, because it is performed after the standard refactoring that has already updated all import declarations.

(S7) PullUp Method "Ship.rotate(int)". The *Pull Up Method* refactoring moves the method `rotate(int)` from class `Ship` to its superclass `SpaceObject`. Our impact analysis detects pseudo alterations of matches, which are moved within the specified hierarchy scope. Since the pull up refactoring never removes members from instances of the sub-class, every instance of `Ship` still contains the method, and thus all pointcut matches remain in the program.

4.5 Experiment 3: Simple Insurance Application

The third experiment underlines the usability of our tool for projects that have properties more comparable to real projects in size, proportion between aspects and classes and the usage of libraries[4]. We refactor an extended version of the *Simple Insurance Application* used in [6].

[4] The project contains 3752 lines of code, defines 59 classes and 3 aspects, and makes use of several libraries.

The program is a scaled down version of an insurance application that keeps track of customers and policies of a fictitious insurance company. The extended version additionally implements a treatment of statistical data for contracted life policies. It defines a new aspect `LifePolicyStatistics` that uses a $cflow$ pointcut and adapts the class `CustomerEditor.AddPolicyListener`. In addition, we have changed the pointcut `findPolicies` of aspect `TrackFinders` to have another pointcut that differs in its characteristics from the other pointcut[5].

```
aspect TrackFinders
findPolicies(String criteria) :
(execution(Set SimpleInsurance.findPoliciesById(
    String))
  || execution(Set SimpleInsurance.
     findPoliciesByCustomerId(String))
  || execution(Set SimpleInsurance.
     findPoliciesByCustomerLastName(String)))
&& args(criteria);

aspect PolicyChangeNotification
notifyingListeners() :
  call(* PolicyImpl.notifyListeners(..))
policyStateUpdate(PolicyImpl policy) :
  execution(* set*(..)) && this(policy)

aspect LifePolicyStatistics
policyContracted() :
  cflow( execution(public void *.widgetSelected(
     SelectionEvent)) )
  && execution(LifePolicyImpl.new(Customer))
```

Figure 6. Pointcuts in Simple Insurance application.

The `TrackFinders` aspect tracks the executions of queries for policies. It defines a pointcut that enumerates the three methods explicitly by specifying their complete signatures and their enclosing type (cf. Figure 6).

The `PolicyChangeNotification` aspect implements a notification mechanism to observe updates of policies. It defines a pointcut to select all executions of *setter* methods of type `PolicyImpl`. The pointcut identifies *setter* methods by the first three characters "set" of their names (cf. Figure 6).

The last aspect `LifePolicyStatistics` implements a treatment of statistical data for contracted life policies. It defines a $cflow$ pointcut to intercept any creation of a `LifePolicyImpl` object "after" the Add-Button was pressed in the user interface (cf. Figure 6). The point in time "after pressing the Add-Button" is specified as "being in the control flow of a method" that is invoked when the Add-Button is pressed. Thus, the pointcut only selects executions of the LifePolicyImpl constructor that are inside the control flow of the method `widgetSelected(SelectionEvent)`.

(S8) Rename Method: "SimpleInsurance.findPoliciesByCustomerLastName(String)". The refactoring renames method `findPoliciesByCustomerLastName(String)` in class `SimpleInsurance` to `findPoliciesByCustomerName`. Our impact analysis reveals one lost match of an exactly specified expression (SQ=100%) in pointcut `findPolicies(String)` of aspect `TrackFinders`. Our tool is able to replace the affected expression and to restore the program behavior.

(S9) Rename Method: "PolicyImpl.createPolicyID()" The *Rename Method* refactoring changes the name of method `createPolicyID()` in class `PolicyImpl` to `setupPolicyID`. Our impact analysis reveals a new, completely scoped match (MS=100%) of expression:

within(subtypes(type("PolicyImpl")),
method(<..>, type(""), "set*", <..>))*

in pointcut `policyStateUpdate(PolicyImpl)` of aspect `PolicyChangeNotification`. The analysis further detects the expression *method(<..>, type("*"), "set*", <..>)* as directly affected by the refactoring *RENAME* transformation. Since this expression is not sufficiently precise (SQ=17%), but quite relevant to the pointcut (ER=75%), the tool proposes to update the pointcut.

The update computation detects other matches when trying to replace the affected expression and proposes to exclude the additional match explicitly:

```
execution(* set*(..)) && this(policy)
&& !execution(private void setupPolicyID())
```

(S10) Inline Local Variable "lp". The *Inline Local Variable* refactoring is applied to the variable `lp` in method `widgetSelected` of class `CustomerEditor.AddPolicyListener`. This refactoring replaces all variable usages with its initialization (cf. lines 281, 283, 286) and affects the pointcut `policyContracted` of aspect `LifePolicyStatistics`. The pointcut captures any instantiation of type `LifePolicyImpl` that occurs in the control flow of any method `widgetSelected(SelectionEvent)`. The cflow property is used to filter instantiations that occur in other contexts. The refactoring duplicates the variable initialization and causes (unintentionally) an alteration of the base program behavior.

Our impact analysis detects a new match path in the call graph between an already existing pair of start- and end-triggers. The additional match path is a new pointcut match with MS=60%. The directly affected expression:

execution(within(type("LifePolicyImpl"),
constructor(<..>, <type("Customer")>)))

is completely specified which leads to SQ=80%. Our tool proposes no update, because the definition of the expression is sufficiently precise and contains an inheritance-based scope (ES=100%). Nonetheless, an altered behavioral property, like cflow, always indicates a changed behavior of the base program, thus, the developer should cancel the refactoring if this alteration is not explicitly intended.

4.6 Discussion

As the primary result, our refactoring tool provided the correct update decisions (and adjustments if proposed) in 9 of 10 refactoring scenarios. In three scenarios (S1, S2, S8), the affected expressions were directly replaced, whereas in scenario S9 the pointcut was extended with an explicit exclusion. Also, in two scenarios (S4, S5) new matches of intentionally under-specified expressions were correctly recognized. However, structurally similar matches with a completely different behavior as in S5 seem not to be recognizable with the current approach.

In Table 2 we present an overview of the experiment, enumerating every refactoring scenario in these experiments. Regarding our two heuristics (specification quality, expression relevance) the refactoring scenarios have dealt with four kinds of pointcuts:

Precise and relevant expressions. For these pointcuts, all lost matches were detected and affected expressions were directly replaced (cf. S1, S2, S3, S8). Hence, as long as a refactoring does not try to remove a matching element, those pointcuts are the most suitable for reliable refactoring support. Since these expressions do not specify dynamic anchors (i.e., ER = high) neither new nor lost matches can be accepted. In particular because of their precise

[5] For a more detailed description of the application cf. [6]. The extended version is available from http://user.cs.tu-berlin.de/˜jwloka/sia.zip.

specification any adjustment keeps the pointcut recognizable even after multiple updates.

Imprecise but relevant expressions. Imprecisely specified properties often cause new matches which represent a special challenge for refactoring tools (cf. S4, S5, S9). For newly matching elements, the refactoring tool has to determine whether they are specified completely, or if they just match accidently. Our approach measures the name-based completeness (SC) and the specified scope of any match (MS) for this decision. Since this kind of expressions is relevant for the evaluation of other parts of the pointcut, we only accept matches of completely unspecified properties (SQ=0%). Our heuristics are able to deal with *intentionally* underspecified matches, however, they cannot deal with poorly written pointcuts in general. If a pointcut is not properly defined our tool warns the developer of altered matches of a poorly written pointcut expression, but it may present a wrong suggestion.

Imprecise and less relevant expressions. Such expressions should only refer to elements which are closely connected to the targeted behavior, because any change can cause new and lost matches, and no analysis of the pointcut can determine whether such matches are intended. For example, in scenario S5, the refactoring created an element with similar properties but the affected pointcut specified the properties incompletely. The result, a nearly correct match, cannot be recognized as a different match because its similarity to the correct matches was not intended. Such expressions are most challenging for aspect-oriented refactoring tools, because the intention whether their matches are wanted remains in the developer's mind.

Precise but less relevant expressions. Our refactoring tool was able to accept the maximal impact for such expressions. The moved method in scenario S7 has even no effect on the selected joinpoint set. The almost completely specified cflow property in scenario S10 also belongs to this category. The only downside is that it is more expensive to differentiate additional occurrences of already specified behavior from alterations of the specified behavior introduced by the refactoring.

5. Related Work

Various approaches have been proposed to cope with evolution issues in aspect-oriented programs [9, 13, 26]. The most related results provide extensions to object-oriented refactorings, new IDE support for determining altered pointcut matches and more expressive pointcut languages, making a pointcut less coupled to brittle implementation details. This section discusses the refactoring and program analysis related approaches in more detail.

Aspect-oriented Refactoring. New refactorings for aspect-oriented programs have been identified which allow for improvement of object-oriented programs by using AO-modularization, the refactoring of aspect language constructs or the refactoring of base code while existing AO adaptations are preserved. In particular, *Monteiro et al.* have developed a catalogue of new refactorings and bad smells for AspectJ programs [14]. *Ceccato et al.* developed an AOPMigrator [5] which supports the extraction of class members and statements into aspects. They propose a specific refactoring workflow that generates a single pointcut for every extracted program element. *Hannemann et al.* present in [11] also a tool for extracting crosscutting concerns into aspects. The tool supports a specific workflow for migrating design patterns implemented in Java to an AspectJ implementation.

All these refactoring approaches have to cope with base code changes that may affect existing pointcuts. Our approach reveals altered joinpoint selections, proposes suitable pointcut updates and generates rephrased pointcuts. Thus it can be seen as an orthogonal extension to these AO refactoring approaches.

Behavior Preservation in AO Refactoring. *Hanenberg et al.* describe initial ideas on how to treat pointcuts within a refactoring workflow [10]. They extend the refactoring constraints to preserve the number of captured joinpoints, the position of a captured joinpoint within the program's control flow and the information provided at a captured joinpoint for every pointcut. Based on these additional constraints, the previous program behavior is reestablished by extending affected pointcuts. Two problems are not considered in this approach: (i) a pointcut may intentionally capture additional joinpoints after a refactoring and (ii) joinpoints are points in the program execution which have to be statically approximated. *Rura and Lerner* advance the AO-specific refactoring constraints [20]. They present a "pointcut pattern equivalence" constraint, that reduces the rule "each advice must apply at semantically equivalent joinpoints" to a more simple determinable but stronger requirement "signature patterns used in a pointcut must match semantically equivalent program elements". In cases where the signature patterns do not match the same elements as before the refactoring, the patterns are broadened/narrowed in a way that new and lost pattern matches are prevented.

However, additional and lost matches are always excluded or included, which makes, on the one hand, pointcuts unrecognizable and more difficult to read, and, on the other hand, it does not consider the pointcut's meaning. Our approach can be seen as an extension to this work. We explicitly consider the change impact of a refactoring by assigning atomic changes with a distinct impact on a pointcut to referenced program structures (i.e., pointcut expressions). Rura and Lerner's approach is limited to signature pattern matches, which e.g., does not allow to move a program element if its matching pattern is restricted by a certain location (i.e., intersected with a $within$ expression). We distinguish whether a selected program element is a joinpoint shadow or used as a pointcut anchor. With this distinction, we can ensure semantically equivalent matches not only of signature patterns, but also of every program structure used to express a joinpoint property.

Change Impact Analysis. Another possibility to cope with fragile pointcuts is tool support that assesses change effects between two program versions. Such an assessment can be done either by comparing two different program versions or by analyzing the applied changes. An approach that detects differently bound advices by comparing two program versions is presented by *Störzer and Graf* in [23]. The proposed pointcut delta analysis approximates the bound joinpoints for every pointcut and compares the pointcut matches between two program versions. *New*, *lost* and *modified* matches (in terms of match quality) are discovered.

Our change impact analysis extends this approach in several ways. The computed match delta is extended to a complete pointcut selection delta that also contains matches for partial sub-sets of the pointcut, rather than only complete pointcut matches. Moreover, our selection delta entries are directly associated with the responsible change. This direct association allows the automated calculation of pointcut updates. Similar to *Ryder et al.* [18, 21] we divide program edits (performed by a refactoring) into their constituent atomic changes. Every change represents a distinct impact on the program's source. Based on the atomic change model Ryder et al. have developed different change classifications that indicate the likelihood of a change to be failure-inducing.

In contrast to Ryder et al. we do not employ the atomic changes to identify affected unit tests, but affected pointcuts. Our change classification indicates the likelihood that a change invalidates the program structures a pointcut relies on. A combination of both approaches allows us to distinguish changes that cause a repairable impact from changes that result in an irreparable impact. With this distinction a tool is enabled to deny critical transformations while others can be safely performed.

6. Conclusions and Future Work

In this paper we have presented a new approach to the refactoring of aspect-oriented programs. Our approach introduces aspect awareness to standard object-oriented refactorings, providing a meta-model for pointcut representations. Based in our meta-model, we have developed a change impact analysis for pointcuts, which computes and assesses the change effects on pointcuts. The resulting change impact information is used for detecting invalidated pointcuts and for computing minimal invasive adjustments of pointcuts. In the evaluation we have shown that our approach can deal with very different kinds of pointcuts, and discussed which kinds can be considered very fragile in general.

Our aspect-aware refactoring approach differs in two ways from existing refactoring approaches: (i) it preserves the characteristics of a pointcut, so it can be recognized by its developer, and (ii) it guides the developer in deciding whether a change invalidates a pointcut, and thus allows for alterations of the composed program behavior. We have shown as the main contributions of our paper: (i) a meta-model for pointcut representations as a general basis for AOP tool-support, (ii) a change impact analysis for pointcuts based on this meta-model, and (iii) two heuristics for impact assessment to automate the detection and adjustment of invalidated pointcuts.

Our approach has been implemented in a refactoring tool, and evaluated in a feasibility study. In this study we have demonstrated that commonly used pointcuts can be statically evaluated, invalidations detected, and pointcut-preserving adjustments computed.

Our evaluation has shown that under-specified joinpoint properties are challenging, but not impossible to solve. However, the use of under specification combined with behavioral properties can lead to wrongly matching joinpoints which are nearly undetectable. They are recognized by similar structural properties but may exhibit completely different behaviors. Future pointcut languages should provide an extra concept for such combinations or prevent them completely.

Our future work will target more general criteria for refactoring-compliant AOP. The exploration and evaluation of equivalence rules for joinpoint properties need to be automated further. Finally, it would be very interesting to see how aspect-aware refactoring can be integrated with class-to-aspect refactorings to make them more generally applicable.

References

[1] *21st IEEE International Conference on Software Maintenance (ICSM 2005), 25-30 September 2005, Budapest, Hungary*. IEEE Computer Society, 2005.

[2] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhotak, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching to AspectJ. Technical Report abc-2005-1, Programming Tools Group, University of Oxford University, UK; BRICS, Group of Aarhus, Denmark; Sable Research, McGill University, Montreal, Canada, 2005.

[3] Homepage of the AspectJ Programming Language. http://www.eclipse.org/aspectj.

[4] L. Bergmans and M. Akşit. Principles and Design Rationale of Composition Filters. pages 63–95. Addison-Wesley, Boston, 2005.

[5] D. Binkley, M. Ceccato, M. Harman, F. Ricca, and P. Tonella. Automated Refactoring of Object Oriented Code into Aspects. In *ICSM* [1], pages 27–36.

[6] A. Colyer, A. Clement, G. Harley, and M. Webster. *Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools*. The Eclipse Series. Addison Wesley Professional, 2004.

[7] Homepage of the Eclipse Java Development Tools (JDT) Subproject. http://www.eclipse.org/jdt/.

[8] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[9] K. Gybels and J. Brichau. Arranging Language Features for Pattern-based Crosscuts. In Mehmet Akşit, editor, *Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003)*, pages 60–69. ACM Press, March 2003.

[10] S. Hanenberg, C. Oberschulte, and R. Unland. Refactoring of Aspect-Oriented Software. In R. Unland, editor, *NetObjectDays*, volume 2591 of *Lecture Notes in Computer Science*. Springer, 2003.

[11] J. Hannemann, G. Murphy, and G. Kiczales. Role-Based Refactoring of Crosscutting Concerns. In Tarr [25], pages 135–146.

[12] H. Masuhara, G. Kiczales, and C. Dutchyn. Compilation Semantics of Aspect-Oriented Programs. In Ron Cytron and Gary T. Leavens, editors, *FOAL 2002: Foundations of Aspect-Oriented Languages (AOSD-2002)*, pages 17–26, March 2002.

[13] T. Mens, K. Mens, and T. Tourwé. Aspect-Oriented Software Evolution. *ERCIM News*, (58):36–37, July 2004.

[14] M. Monteiro and J. Fernandes. Towards a Catalog of Aspect-Oriented Refactorings. In Tarr [25], pages 111–122.

[15] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. Ph.d. thesis, University of Illinois at Urbana-Champaign, Urbana-Champaign, IL, 1992.

[16] K. Ostermann, M. Mezini, and C. Bockisch. Expressive Pointcuts for Increased Modularity. In Andrew P. Black, editor, *ECOOP*, volume 3586 of *Lecture Notes in Computer Science*, pages 214–240. Springer, 2005.

[17] Online Catalog of Refactorings. http://www.refactoring.com/catalog/index.html.

[18] X. Ren, B. G. Ryder, M. Stoerzer, and F. Tip. Chianti: A Change Impact Analysis Tool for Java Programs. In Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh, editors, *27th International Conference on Software Engineering (ICSE 2005)*, pages 664–665, New York, NY, USA, May 2005. ACM Press.

[19] Research on Object-Oriented Technologies and Systems Department at University of Bonn. *Homepage of the LogicAJ Project*. available from `http://roots.iai.uni-bonn.de/research/logicaj/`.

[20] S. Rura and B. Lerner. A Basis for AspectJ Refactoring. http://www.mtholyoke.edu/ blerner/papers/gpce04.pdf, 2004.

[21] B. G. Ryder and F. Tip. Change Impact Analysis for Object-oriented Programs. In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 46–53, New York, NY, USA, 2001. ACM Press.

[22] Source Code of the Spacewar Example Program. http://dev.eclipse.org/viewcvs/index.cgi/org.eclipse.ajdt/AJDT_src/org.eclipse.ajdt.examples/examples/spacewar/?root=Tools_Project.

[23] M. Störzer and J. Graf. Using Pointcut Delta Analysis to Support Evolution of Aspect-Oriented Software. In *ICSM* [1], pages 653–656.

[24] System and Software Engineering lab (SSEL) at the Department of (Applied) Computer Science (Faculty of Sciences) at Vrije Universiteit Brussel (VUB). *JAsCo Language Reference*. available from `http://ssel.vub.ac.be/jasco/documentation:main`.

[25] Peri Tarr, editor. *Proc. 4rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2005)*. ACM Press, March 2005.

[26] T. Tourwé, J. Brichau, and K. Gybels. On the Existence of the AOSD-Evolution Paradox. In Lodewijk Bergmans, Johan Brichau, Peri Tarr, and Erik Ernst, editors, *SPLAT: Software engineering Properties of Languages for Aspect Technologies*, Boston, USA, March 2003.

[27] J. Wloka. *Tool-supported Refactoring of Aspect-oriented Programs*. Ph.d. thesis, Technical University Berlin, Berlin, Germany, 2007.

[28] Xerox Corporation. *AspectJ Programming Guide*. available from `http://eclipse.org/aspectj`.