Aspect Refactoring Verifier*

Charles Zhang and
Hans-Arno Jacobsen
Department of Electrical and Computer Engineering
and Department of Computer Science
University of Toronto
{czhang,jacobsen}@eecg.toronto.edu

Julie Waterhouse
Centers for Advanced Studies
IBM Toronto Lab
juliew@ca.ibm.com

Adrian Colyer
IBM Hursley Lab
adrian_colyer@uk.ibm.com

Keywords
Aspect-Oriented Programming, Aspect Verification, Aspect Mining

1. INTRODUCTION

When performing refactoring, the principal requirement is that the refactored code should be at least functionally equivalent[1] to the original. While execution tests provide the authoritative verification of this functional equivalence, verification at the source level is often more effective because mistakes can be detected early and fixed as part of the development activity. Automatic refactoring performed by development tools such as Eclipse[2] typically provides this kind of source-level verification. However, its capabilities are currently limited to changing hierarchical structures involving methods and classes.

Aspect-oriented refactoring fundamentally differs from traditional refactoring because it typically involves multiple elements across the decomposition hierarchy. In order for automatic refactoring of aspects to be feasible in a large software code base, tool support for source-level verification is necessary for two reasons. First, a manual refactoring process is tedious and error-prone. For instance, our refactoring project targeting a mid-size middleware implementation consists of 61 aspects and 243 pointcuts, which potentially affect or "advise" 273 places. The possible locations in the code that are affected by the aspects are known as "shadows". These shadows span over 14 packages and 103 methods. The sheer quantity and diversity of the affected locations show that the verification would be better tackled with tool support. Second, the constructs of aspect-oriented languages are dramatically different from those of traditional languages. Compared to conventional refactoring, aspect-oriented refactoring does not use a consistent set of language elements, which makes manual comparison even more difficult. One way this difficulty manifests itself is that, when refactoring a method call pertaining to a crosscutting concern, the refactored code needs to re-enact the original call flow. It must do this by capturing the calling context of the method call (i.e., its caller or control flow information), rather than the method call itself. At the verification stage, it becomes a comparison of apples and oranges, likely leading to confusion. Another example of this kind of verification difficulty arises when a pointcut pattern is used to capture a group of places to be refactored. Patterns do not discriminate when matching program elements; they can include not only the intended places, but also potentially unintended places.

The Aspect Refactoring Verification tool (ARV) is an Eclipse plug-in we have built that works in conjunction with AJDT[3], the Eclipse AspectJ development environment. ARV is a first step towards automatically verifying refactored aspects against the original source. ARV supports an integrated refactoring process where the aspect mining or the aspect exploration information is available as the base reference for verification. Figure 1 illustrates this integrated refactoring process and where ARV fits in. In this process, aspect discovery techniques are initially employed to identify the locations, or "footprints", of aspects in legacy applications. Aspect footprints serve as guidance for either automatic or manual refactoring of tangled code into aspects. The verification is then carried out to capture any inequalities between the refactored code and the original sources. Such inequalities would most likely lead to a failure to preserve the original functionality. The behavioural equivalence is evaluated as the last stage of the verification process, through unit and integration tests.

In general, a comparison between the refactored code and the original code can lead to three possible results:

*In Linking Aspect Technology and Evolution Workshop, AOSD 2005, Chicago, IL, USA

[1]We refer to this functional equivalence as the preservation of the functional behaviour of the system according to a set of well-defined measurements.
[2]Eclipse. URL:`http://www.eclipse.org`

[3]Eclipse AspectJ Development Tools. URL:`http://www.eclipse.org/ajdt`

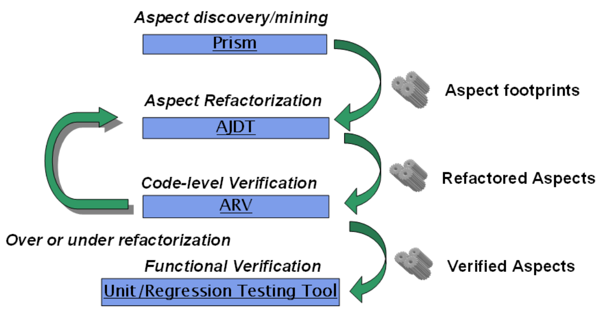


Figure 1: Integrated Mining - Refactoring - Verification Process

1. Equivalence: This is the desired result, whereby the original and refactored systems are equivalent with respect to the code affected by the refactored aspect.

2. Under-Refactoring: In this case, the tool allows the user to easily detect that the aspect has matched too few places in the tangled code.

3. Over-Refactoring: Here, the tool reveals places in the refactored code where the refactored implementation is providing advice, but the tangled logic was not indicating a need for such advice. In our experience, most of the time that this situation occurs, the aspect is right, and the comparison is highlighting a bug in the original program. Whether or not the original program is correct, the user needs to be made aware of the difference so that it can be investigated and appropriate action taken.

It is commonly considered difficult to provide equivalence verification based on source code. However, we believe that verification is viable for certain specialized refactoring scenarios, and that functional non-equivalence, including both under-refactoring and over-refactoring, is often much easier to detect than equivalence. Such detection is helpful for developers in refactoring large software systems. The current implementation of ARV focuses on comparing the code locations advised by refactored aspects with the location information gathered at the aspect discovery stage, before the refactoring starts. We approximate, but do not guarantee, logic equivalence between the refactored code and the original source, in terms of affected locations.

In the rest of the paper, section 2 gives a detailed description of both the interface and the verification method implemented in ARV. Section 3 presents a use case of ARV that illustrates both under- and over-refactoring scenarios.

2. ASPECT REFACTORING VERIFIER

ARV is integrated with the AspectJ Development Tools (AJDT) Eclipse Plug-in and can be accessed through two views: the "ARV Refactoring Verifying" view and the "Verification Results" view. These two views must be explicitly opened as in Figure 2a. Currently, a specific AJDT project must be selected in ARV before it can be analyzed as in Figure 2b. ARV collects data about pointcuts and their shadows in the code while AJDT builds the project. The comparison functionality is available after the build of the project completes. Figure 3 shows the verifier in action.

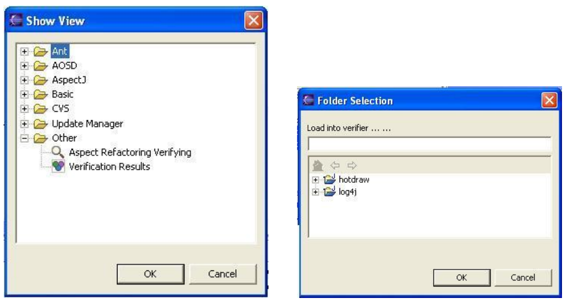


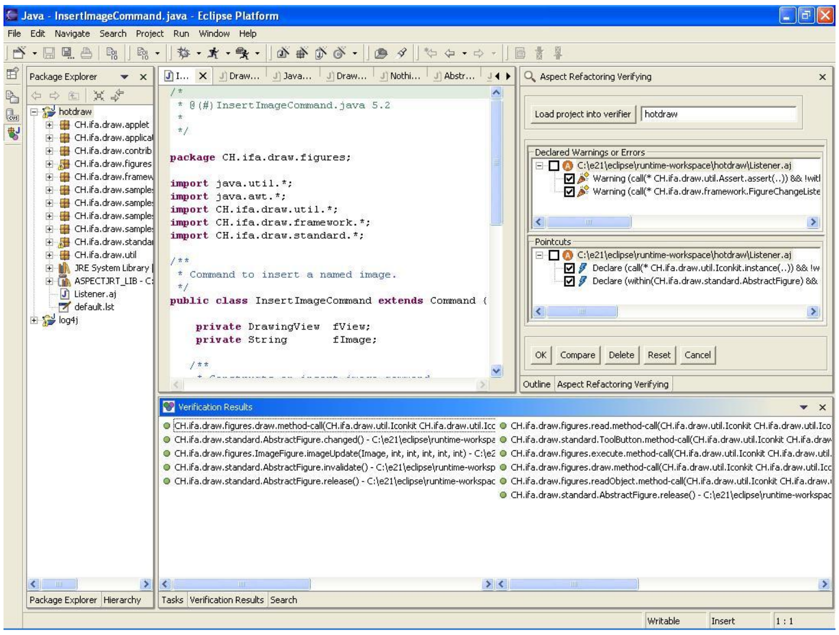
Figure 2: a. Open ARV Views b. Select Project

ARV makes use of mechanisms in the AspectJ language to compare the source code locations found during the aspect search and discovery stage with those that would be affected by the proposed aspect refactoring. It is common practice that during the discovery stage, "`declare warning`," an AspectJ language construct, is used to search for crosscutting concerns. This mechanism uses type patterns in conjunction with AspectJ pointcuts. Call-based pointcuts are the most common means of expressing the search, but other pointcut designators, such as `handler`, may also be used. Once the target locations that represent the crosscutting concern have been identified, they need to be compared to the actual locations that will be affected by the aspect written to encapsulate the equivalent behaviour in the refactored code. Any of the AspectJ pointcut expressions may be used to capture the source locations that may be advised by the new aspect. ARV provides a visual comparison of the places matched by the "`declare warnings`" with the places advised by the aspect. This is accomplished by extracting the shadows collected by the AspectJ compiler for both the "`declare warnings`" and the pointcut definitions in the aspect.

3. AN ARV USE CASE

Let us illustrate the use of ARV through an aspect-oriented refactoring effort performed on JHotdraw[4], an open source Java^{TM}GUI framework. Our goal in this example is to refactor two functionalities: observers and assertions. They are widely considered to be crosscutting concerns and better modularized in aspect modules. For illustration purposes, we have manually injected some assertion calls into the code base which re-enforce certain existing validity-checking policies in the original application. Our refactoring process starts with the use of the AspectJ construct "`declare warning`" for capturing where observers and assertions occur in the code base, as shown by the code snippet in Figure 4. Alternatively, an aspect-mining tool such as Prism [4] could be used for this purpose. Here, for simplicity, we only capture the calls to a particular listener `FigureChangeListener`. Figure 5 shows the AJDT crosscutting view (right) of this pointcut and one instance of captured calls (left).

[4]JHotdraw URL:`www.jhotdraw.org`


Java - InsertImageCommand.java - Eclipse Platform
File Edit Navigate Search Project Run Window Help
Package Explorer
hotdraw
CH.ifa.draw.applet
CH.ifa.draw.applicat
CH.ifa.draw.contrib
CH.ifa.draw.figures
CH.ifa.draw.framew
CH.ifa.draw.sample:
CH.ifa.draw.sample:
CH.ifa.draw.sample:
CH.ifa.draw.sample:
CH.ifa.draw.standar
CH.ifa.draw.util
JRE System Library [
ASPECTJRT_LIB - C:
Listener.aj
default.lst
log4j
I... Draw... Java... Draw... Nothi... Abstr...
Aspect Refactoring Verifying
Load project into verifier
hotdraw
Declared Warnings or Errors
Pointcuts
OK Compare Delete Reset Cancel
Outline Aspect Refactoring Verifying
Verification Results
Package Explorer Hierarchy
Tasks Verification Results Search
Writable Insert 1 : 1

```
/*
 * @(#)InsertImageCommand.java 5.2
 *
 */

package CH.ifa.draw.figures;

import java.util.*;
import java.awt.*;
import CH.ifa.draw.util.*;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.standard.*;

/**
 * Command to insert a named image.
 */
public class InsertImageCommand extends Command {

    private DrawingView  fView;
    private String       fImage;

   /**
```

Declared Warnings or Errors:

```
C:\e21\eclipse\runtime-workspace\hotdraw\Listener.aj
    Warning (call(* CH.ifa.draw.util.Assert.assert(..)) && !witl
    Warning (call(* CH.ifa.draw.framework.FigureChangeListe
```

Pointcuts:

```
C:\e21\eclipse\runtime-workspace\hotdraw\Listener.aj
    Declare (call(* CH.ifa.draw.util.Iconkit.instance(..)) && !w
    Declare (within(CH.ifa.draw.standard.AbstractFigure) &&
```

Verification Results:

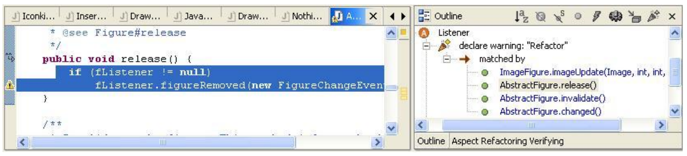
```
CH.ifa.draw.figures.draw.method-call(CH.ifa.draw.util.Iconkit CH.ifa.draw.util.Icc   CH.ifa.draw.figures.read.method-call(CH.ifa.draw.util.Iconkit CH.ifa.draw.util.Ico
CH.ifa.draw.standard.AbstractFigure.changed() - C:\e21\eclipse\runtime-workspa    CH.ifa.draw.standard.ToolButton.method-call(CH.ifa.draw.util.Iconkit CH.ifa.draw
CH.ifa.draw.figures.ImageFigure.imageUpdate(Image, int, int, int, int, int) - C:\e2 CH.ifa.draw.figures.execute.method-call(CH.ifa.draw.util.Iconkit CH.ifa.draw.util.
CH.ifa.draw.standard.AbstractFigure.invalidate() - C:\e21\eclipse\runtime-worksp   CH.ifa.draw.figures.draw.method-call(CH.ifa.draw.util.Iconkit CH.ifa.draw.util.Icc
CH.ifa.draw.standard.AbstractFigure.release() - C:\e21\eclipse\runtime-workspac    CH.ifa.draw.figures.readObject.method-call(CH.ifa.draw.util.Iconkit CH.ifa.draw.
                                                                                   CH.ifa.draw.standard.AbstractFigure.release() - C:\e21\eclipse\runtime-workspac
```

Figure 3: The Aspect Refactoring Verifier

```
1. declare warning:
2. call(* FigureChangeListener.*(..))&&
3. !within(FigureChangeListener+)
4. &&!within(MyAspect):"Refactor";

5. declare warning:
6. call(* Assert.assert(..))&&
7. !within(MyAspect):"Condition check";
```

Figure 4: Declare warnings


Iconki... Inser... Draw... Java... Draw... Nothi... A...
Outline
Outline Aspect Refactoring Verifying

```
   * @see Figure#release
   */
  public void release() {
      if (fListener != null)
          fListener.figureRemoved(new FigureChangeEven
  }

  /**
```

```
Listener
  declare warning: "Refactor"
    matched by
      ImageFigure.imageUpdate(Image, int, int,
      AbstractFigure.release()
      AbstractFigure.invalidate()
      AbstractFigure.changed()
```

Figure 5: Disclosed code by Declare-Warnings

```
after(AbstractFigure f):
within(AbstractFigure)&&
execution(* AbstractFigure.release())&&target(f)
{
  if (f.fListener != null)
     f.fListener.figureRemoved(
     new FigureChangeEvent(f));
}
```

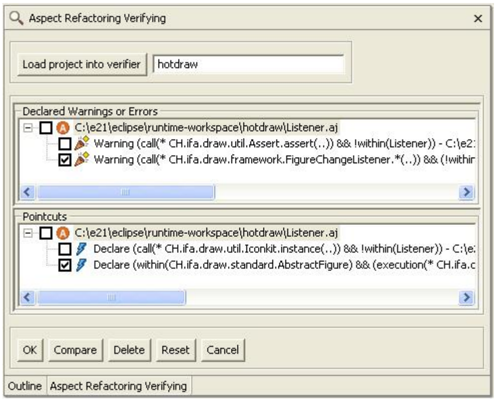
Figure 6: Refactor into aspects

3.1 Under-refactoring

Under-refactoring means that, when the code is refactored into aspects, not all of the original functionality is preserved. This can happen because the joinpoints in the refactored aspects need to be composed in a different manner from the joinpoints used in "`declare warning`," and so it is not easy to be sure that they are equivalent. Using our listener example, the legacy code captured by the "`call`"-based joinpoint in "`declare warning`"(Figure 4, line 2) must be rewritten as "`execution`"-based joinpoints in code snippet 6. In this simple example, an under-refactoring occurs since the aspect code only refactors one of the four places captured using "`declare warning`". The under-refactored places can be easily shown in ARV by selecting both the "`declare warning`" pointcut and the refactored pointcut and clicking the "compare" button, as illustrated in Figure 7. The "Verification Results" view initially shows all captured places of both sides, as in Figure 8. Selecting the "Show under-refactored" option then shows on the left pane the three places that still need to be refactored, as illustrated in Figure 9. Though these steps are unnecessary in our simple scenario, for which the mismatch is obvious, the number of matched places can be considerable in a large-scale refactoring, and could be tedious to verify manually.

3.2 Over-refactoring

Over-refactoring means that the refactored aspect extends the original functionality by injecting it at more locations in the code base than were present in the pre-refactored version. To illustrate over-refactoring, we made a minor modification to the original code, as shown in Figure 10, which uses a generic assertion to replace the direct use of exceptions. The original code (top part) represents a validating policy enforcing the checking of the "nullness" of the singleton `Iconkit`. This validation policy can be nicely captured in AspectJ, as shown in Figure 11. Following the same steps as the previous example, but this time selecting "Show over-refactored," ARV outputs four over-refactored places, as illustrated in Figure 12. Double-clicking one of the reported locations opens the code editor and reveals the inconsistency in the enforcement of this validation policy in the original code, as shown in Figure 13.

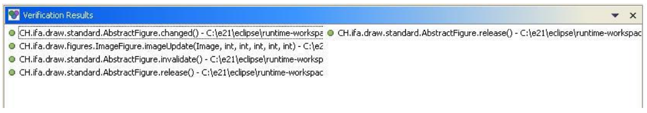


```
hotdraw

Declared Warnings or Errors
⊟ ☐ C:\e21\eclipse\runtime-workspace\hotdraw\Listener.aj
    ☐ Warning (call(* CH.ifa.draw.util.Assert.assert(..)) && !within(Listener)) - C:\e2
    ☑ Warning (call(* CH.ifa.draw.framework.FigureChangeListener.*(..)) && (!withir

Pointcuts
⊟ ☐ C:\e21\eclipse\runtime-workspace\hotdraw\Listener.aj
    ☐ Declare (call(* CH.ifa.draw.util.Iconkit.instance(..)) && !within(Listener)) - C:\e
    ☑ Declare (within(CH.ifa.draw.standard.AbstractFigure) && (execution(* CH.ifa.c
```

Figure 7: Selecting Pointcuts for Comparison



```
● CH.ifa.draw.standard.AbstractFigure.changed() - C:\e21\eclipse\runtime-workspa   ● CH.ifa.draw.standard.AbstractFigure.release() - C:\e21\eclipse\runtime-workspac
● CH.ifa.draw.figures.ImageFigure.imageUpdate(Image, int, int, int, int, int) - C:\e2
● CH.ifa.draw.standard.AbstractFigure.invalidate() - C:\e21\eclipse\runtime-worksp
● CH.ifa.draw.standard.AbstractFigure.release() - C:\e21\eclipse\runtime-workspac
```

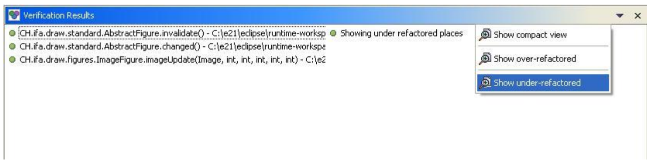
Figure 8: Results of Comparison

3.3 Current Limitation

Both over- and under-refactoring are computed by filtering out identical call sites captured by pointcuts from both sides of the comparison. Currently, the test for equality between two call sites is evaluated according to whether the sites affect the same method. The exact locations of the call sites cannot be used if call-based "`declare warnings`" are to be compared with the refactored "execution"-based pointcuts. These two types of joinpoints typically have different shadows, i.e., different line positions. ARV, in addition to the method name, also displays the exact locations of the call sites for further verification. Although this verification is done manually, ARV narrows the number of matches and provides views to facilitate direct comparison.

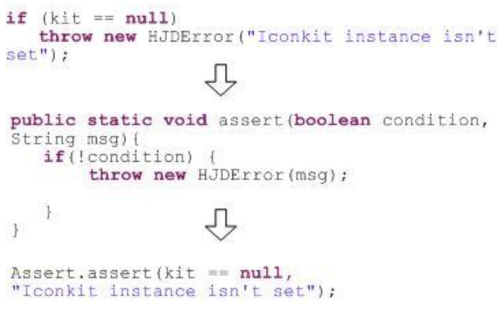
4. RELATED WORK

ARV is intended to be a tool for capturing the crosscutting difference between an original and a refactored program. In that sense, ARV has similar objectives to tools identifying the difference between two input files, like the common UNIX®diff tool. The current implementation of ARV resembles that of PCDiff [3], which also compares differences between sets of AspectJ pointcut shadows. The primary functionality of PCDiff is to automatically track changes in both base code and aspects to guard against the fragility of AspectJ pointcut definitions. The direction of ARV, however, is to compare plain Java source to its aspect-refactored



```
● CH.ifa.draw.standard.AbstractFigure.invalidate() - C:\e21\eclipse\runtime-worksp   ● Showing under refactored places
● CH.ifa.draw.standard.AbstractFigure.changed() - C:\e21\eclipse\runtime-workspa
● CH.ifa.draw.figures.ImageFigure.imageUpdate(Image, int, int, int, int, int) - C:\e2
```

Figure 9: Showing under-refactored places



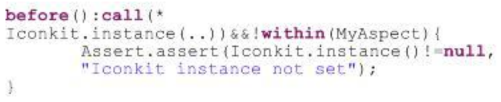
```
if (kit == null)
    throw new HJDError("Iconkit instance isn't
set");
```

```
public static void assert(boolean condition,
String msg){
    if(!condition) {
        throw new HJDError(msg);

    }
}
```

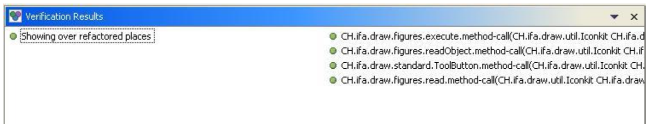
```
Assert.assert(kit == null,
"Iconkit instance isn't set");
```

Figure 10: Rewriting Exception Handling in Assertion



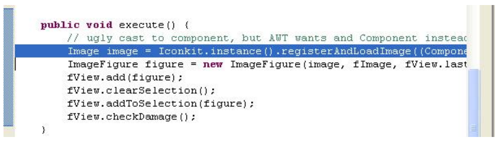
```
before():call(*
Iconkit.instance(..))&&!within(MyAspect){
        Assert.assert(Iconkit.instance()!=null,
        "Iconkit instance not set");
}
```

Figure 11: Enforcing Validation in AspectJ



```
● Showing over refactored places          ● CH.ifa.draw.figures.execute.method-call(CH.ifa.draw.util.Iconkit CH.ifa.d
                                          ● CH.ifa.draw.figures.readObject.method-call(CH.ifa.draw.util.Iconkit CH.if
                                          ● CH.ifa.draw.standard.ToolButton.method-call(CH.ifa.draw.util.Iconkit CH.
                                          ● CH.ifa.draw.figures.read.method-call(CH.ifa.draw.util.Iconkit CH.ifa.draw
```

Figure 12: Showing Over-refactoring



```
public void execute() {
    // ugly cast to component, but AWT wants and Component insteac
    Image image = Iconkit.instance().registerAndLoadImage((Compone
    ImageFigure figure = new ImageFigure(image, fImage, fView.last
    fView.add(figure);
    fView.clearSelection();
    fView.addToSelection(figure);
    fView.checkDamage();
}
```

Figure 13: Over-refactoring Reveals Inconsistency in Original Code

equivalent. The use of "`declare warning`" is an interim means to express the user's refactoring intent.

In current aspect-oriented refactoring approaches, rigorous principles and algorithms are applied to provide some degree of guarantee of the correctness of certain types of refactored code. Ettinger and Verbaere [1] propose to use aspects to refactor program slices. They have built a refactoring tool, Nate, to support the static slicing of programs in a small subset of the Java language through the use of a demand-driven, inter-procedural slicing algorithm. The ART (Aspect-Oriented Refactoring Tool) tool [2] aims at leveraging program dependence graphs (PDGs) to enable certain types of refactoring. Refactoring based on slicing techniques or PDGs can algorithmically ensure the equivalence between the original source and the refactored code.

5. CONCLUSION

The goal of ARV is to help the programmer ensure completeness and correctness when using aspect-oriented programming to refactor large legacy systems. The current implementation of ARV focuses on comparing call sites, i.e., comparing the call sites captured through "`declare warning`" constructs, with the call sites affected by the refactored aspects. In addition to displaying call sites for both sides of the comparison, ARV provides two additional perspectives. The under-refactoring view lists all call sites that are intended, but fail, to be refactored. This typically implies programming errors in the refactoring process. The over-refactoring view lists new call sites in the legacy code that are affected by refactored aspects. These call sites typically imply bugs in the original code.

The ongoing focus of ARV is the effective verification of refactored aspects at the code level. For certain refactoring cases, it will also be possible for ARV to ensure the advice body is an equivalent transformation of the original code, and that it applies at the correct places, i.e., before, after, or around the original methods.

Acknowledgments

This research has been supported in part by an NSERC[5] grant and in part by an IBM CAS Fellowship for the first author. Irum Godil has made significant contributions to the development of ARV. The authors are very grateful for the support.

The views expressed in this paper are those of the authors and not necessarily of IBM Canada Ltd. or IBM Corporation.

[5]Natural Sciences and Engineering Research Council of Canada

6. REFERENCES

[1] Ran Ettinger and Mathieu Verbaere. Untangling: a slice extraction refactoring. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 93–101. ACM Press, 2004.

[2] M. Iwamoto and J. Zhao. Refactoring aspect-oriented programs. In *4th AOSD Modeling With UML Workshop, UML'2003, San Francisco, California, USA, October 20, 2003.*

[3] Christian Koppen and Maximilian Stoerzer. PCDiff: Attacking the Fragile Pointcut Problem. In *European Interactive Workshop on Aspects in Software (EIWAS 04)*, September 2004.

[4] Charles Zhang and Hans-Arno Jacobsen. Prism is research in aspect mining. In *Companion of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications.* ACM Press, 2004.