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
The basic guarantee when performing refactoring is that the refactored code should be at least functionally equivalent to the original code. 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[1] typically provide such guarantees. However, these automatic refactoring 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 hierarchical structure. Before automatic refactoring of aspects comes to reality, tool support for source-level verification becomes necessary in dealing with large software code bases for two reasons: A. Manual 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 (shadows). These shadows span over 14 packages and 103 methods. The sheer quantity and diversity of the locations show that the verification is rather better tackled with tool support; B. The constructs of aspect oriented languages are dramatically different from that of traditional languages. Compared to conventional refactoring, aspect oriented refactoring does not use the same set of language elements which make manual comparison even more difficult. For instance, to refactor a method call pertaining to a crosscutting concern, the refactored code needs to reenact the original call flow by capturing the calling context of the method call, i.e., its caller information or its 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 kind of verification difficulty arises when a pointcut pattern is used to capture a group of to-be-refactored places. Patterns are non-discriminating to all matching program elements including not only the intended places but also potentially unintended places. It is also difficult to manually verify this effect.

The aspect refactoring verification tool (ARV) is an Eclipse plugin we have built working in conjunction with AJDT[2], the Eclipse AspectJ development environment. ARV is a first step towards automatically verifying refactored aspects against the original source. ARV supports the 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 locations or the "footprints" of aspects in legacy applications. Aspect footprints serve as the guidance for either automatic or manual refactoring of tangled code into aspects. The verification is then carried out primarily to capture the inequalities between the refactored code and the original sources. These inequalities are most likely leading to the failure of the preservation of the original functionality. The authoritative verification is performed at the last stage through unit and integration tests.

"Declare warning" is an AspectJ language construct can be and is used to explore or search crosscutting concerns through AspectJ pointcut facilities and type patterns. Using call based pointcuts to express the search is one common means, but other pointcut designators may also be used such as `handler`. Likewise the places advised by the aspect may use any pointcut expression as needed to capture the concern. Currently, ARV performs a comparison of the places matched by the declare warning (a call-based pointcut) with the places advised by the aspect (an execution based pointcut). The verification must consider whether the original

*MSRG Technical Communication, University of Toronto
[1]Eclipse. URL:`http://www.eclipse.org`

[2]Eclipse AspectJ Development Tool. URL:`http://www.eclipse.org/ajdt`

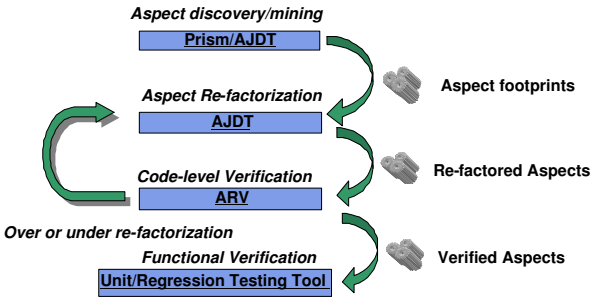

Figure 1: Integrated Mining - Refactoring - Verification Process

and the refactored program are functionally equivalent - which involves a comparison of both the places advised, and also the logic contained in the advice.

This comparison can lead to three possible results:

1. Equivalence: This is the desired result, original and refactored systems are equivalent with respect to the refactored aspect.
2. Under-Refactoring: This would allow the user to easily see places in the code that the aspect does not match the tangled logic.
3. Over-Refactoring: This shows 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 when this situation occurs, the aspect is right and the comparison is highlighting a bug in the original program. In either case, the user needs to be made aware of the difference so that it can be investigated and appropriate action taken.

ARV is integrated with the AspectJ Development Tool Eclipse Plugin (AJDT) and can be accessed through two views: the ARV specification view and the difference view. These two views must be explicitly opened as in Figure 2 (a). Currently, a specific AJDT project must be selected (Figure 2 (b)) in ARV before it can be analyzed. ARV collects data about pointcuts and their affecting locations in the code while AJDT builds the project. The comparison functionality is available after the project is built by AJDT. Figure 3 shows the verifier in action.

2. ASPECT REFACTORING VERIFICAITON

Let us illustrate of the use of ARV through an aspect oriented refactoring effort performed on JHotdraw[3], an open source Java GUI framework. Our goal in this example is to refactor two functionalities: observers and assertions. They are widely considered as crosscutting concerns and better modularized in aspect modules. For illustration purposes,

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

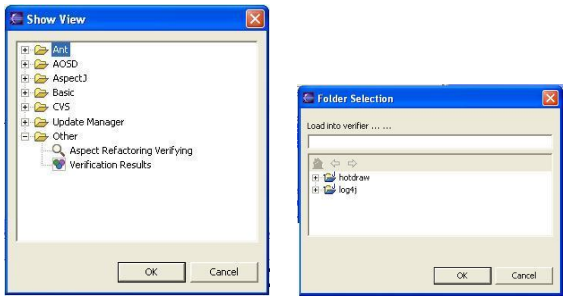

Figure 2: a. Open ARV Views b. Select Project

```
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

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 in the code snippet 4. Alternatively, an aspect mining tool such as Prism [3] could be used for this purpose. For simplicity, we here 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).

2.1 Under-refactoring

Under-refactoring means that, when refactored into aspects, the original functionality is insufficiently preserved. This can happen because the joinpoints in the refactored aspects need to be composed differently than the joinpoints used in "declare warning". 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, a under-refactoring occurs since the aspect code only refactors one of the four places captured using "declare warning". And under-refactored places can be easily shown in ARV by firstly selecting both the "decare warning" poincut and the refactored pointcut and clicking the "compare" button, 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 shows

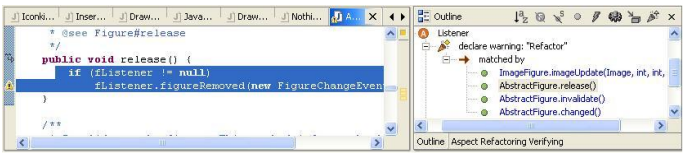

Figure 5: Disclosed code by Declare-Warnings

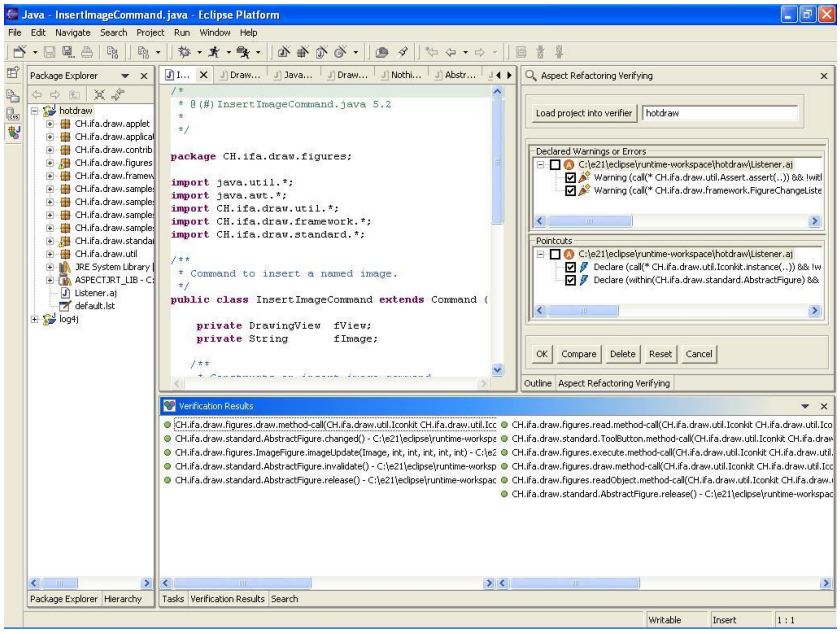

Figure 3: The Aspect Refactoring Verifier

```
1. after(AbstractFigure f):
2. within(AbstractFigure)&&
3. execution(* AbstractFigure.release())&&target(f)
4. {
5.   if (f.fListener != null)
6.       f.fListener.figureRemoved(
7.       new FigureChangeEvent(f));
8. }
```

Figure 6: Refactor into aspects

on the left pane the three places still need to be refactored as illustrated in Figure 9. Though obvious and unnecessary in our simple scenario, the number of matched places can grow extremely large in the large scale refactoring and become tedious to verify manually.

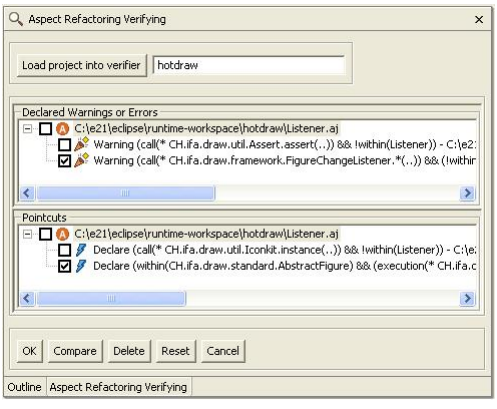

Figure 7: Selecting Pointcuts for Comparison

2.2 Over-refactoring

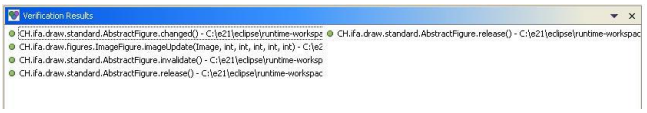

Figure 8: Results of Comparison

Figure 9: Showing under-refactored places

Over-refactoring means the refactored aspects extends the original functionality by injecting it at more locations in the code base as compared to 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 in Figure 11. Following the same steps as the previous example, by selecting "Show over-refactored" ARV outputs four over-refactored places illustrated in Figure 12. Double clicking one of the reported locations opens the code editor and reveals the consistency of enforcing this validation policy in the original code as show in Figure 13.

2.3 Current Limitation

Both over and under refactoring are computed by filtering out identical call sites captured by pointcuts from both sides

```
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

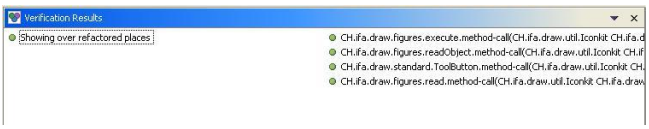

Figure 12: Showing Over-refactoring

```
public void execute() {
    // ugly cast to component, but AWT wants and Component instead
    Image image = Iconkit.instance().registerAndLoadImage((Compon
    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

of the comparison. Currently, the test for the equality between two call site are evaluated as whether they affect the same method. The exact locations of the call sites cannot be used since "declare warning" and the refactored "execution" based pointcuts typically have different pointcut 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 direct views to facilitate such verifications.

3. RELATED WORK

We are not aware of past or current project that are immediately related to the ARV project described in this paper. ARV intends to be a tool to capture the corsscutting difference between original and refactored program. In that sense, ARV has similar objectives to tools identifying the difference between two input files, like the common UNIX diff tool. In current aspect oriented refactoring approaches, rigorous principles and refactoring algorithms are applied to provide a certain degree of guarantees for the correctness of the 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 a small subset of Java through the use of a demand-driven inter-procedural slicing algorithm. The ART tool [2] aims at leveraging program dependence graphs(PDG) 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.

4. CONCLUSION

The goal of ARV is to help ensuring of the completeness and the correctness of using AOP to refactor large legacy systems. The current implementation of ARV focuses on comparing call sites, i.e., comparing call sites captured through the "declare warning" constructs with the affected call sites by the refactored aspects. In addition to listing call sites for both sides of the comparison, ARV provides two additional perspectives. The under-refactoring view lists all call sites which are intended but failed to be refactored. This typically means 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 mean bugs in the original code.

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

Acknowledgments

This research has been supported in part by an NSERC grant and in part by an IBM CAS fellowship for the first author. The authors are very grateful for this support.

5. 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. 4th AOSD Modeling With UML Workshop, UML'2003, San Francisco, California, USA, October 20, 2003.

[3] 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.