UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled “Semi-automating Small-Scale Source Code Reuse via Structural Correspondence” submitted by Rylan Cottrell in partial fulfillment of the requirements for the degree of Master of Science.

______________________________
Supervisor / Examining Committee Member
Dr. Robert J. Walker
Department of Computer Science

______________________________
Co-supervisor / Acting Supervisor
Dr. Jörg Denzinger
Department of Computer Science

______________________________
Examiner Committee Member
Dr. Jonathan Sillito
Department of Computer Science

______________________________
Examiner Committee Member
Dr. Hude Quan
Department of Community Health Sciences

August 28, 2008
Date
Abstract

Developers perform small-scale reuse tasks to save time and to increase the quality of their code, but due to their small scale, the costs of such tasks can quickly outweigh their benefits. Existing approaches focus on locating source code for reuse but do not support the integration of the located code within the developer’s system, thereby leaving the developer with the burden of performing integration manually. This thesis presents an approach that uses the developer’s context to help integrate the reused source code into the developer’s target source code. The approach approximates a theoretical framework (higher-order anti-unification modulo theories), known to be undecidable in general, to determine candidate correspondences between the source code to be reused and the developer’s current (incomplete) system. This approach has been implemented in a prototype tool, called Jigsaw, that identifies and evaluates candidate correspondences greedily with respect to the highest similarity. Situations involving multiple candidate correspondences with similarities above a defined threshold are presented to the developer for resolution. Two empirical evaluations were conducted: an experiment comparing the quality of Jigsaw’s results against suspected cases of small-scale reuse in an industrial system; and case studies with two industrial developers to consider its practical usefulness and usability issues.
# Table of Contents

Approval Page .................................................. ii  
Abstract .......................................................... iii  
Table of Contents ................................................ iv  
List of Tables ................................................... vi  
List of Figures ................................................... vii  
1 Introduction ................................................... 1  
1.1 Background of small-scale reuse ............................ 3  
   1.1.1 Overview of related work ............................... 4  
1.2 Broad thesis overview ..................................... 4  
1.3 Thesis statement .......................................... 5  
1.4 Structure of thesis ....................................... 5  
2 Motivational Scenario ....................................... 7  
2.1 Summary ................................................... 12  
3 Theoretical foundations ..................................... 13  
3.1 Structure ................................................... 13  
3.2 Anti-unification .......................................... 13  
3.3 Higher-order anti-unification modulo theories ............ 17  
3.4 Abstract syntax tree ...................................... 21  
3.5 Summary ................................................... 25  
4 The Jigsaw Tool ............................................... 26  
4.1 Running example .......................................... 30  
4.2 Constructing the CAST .................................... 31  
4.3 Anti-unifier of best fit ................................... 39  
4.4 Relevance links .......................................... 42  
4.5 Integration ................................................ 43  
4.6 Semantics ................................................ 47  
4.7 Multiple MSAs ............................................ 48  
4.8 Summary ................................................... 50  
5 Evaluation ..................................................... 51  
5.1 Experiment ............................................... 51  
   5.1.1 Results ............................................... 53  
   5.1.2 Lessons learned ....................................... 55  
5.2 Case studies ............................................... 57  
   5.2.1 Results ............................................... 57  
5.3 Summary ................................................... 58  
6 Related Work .................................................. 60  
6.1 Reuse ...................................................... 60  
   6.1.1 Locating examples of reuse ............................ 60  
   6.1.2 Integration ............................................ 61  
6.2 Copy & paste ............................................. 62  
6.3 Correspondence .......................................... 63
List of Tables

5.1 Method pairs used as test cases; all are contained in the org.eclipse.jdt package. .......................................................... 52
5.2 Results from applying Jigsaw to the 15 (+ 1) test cases. Times marked thus (†) do not include the time required for interaction with the developer in cases where developer input was needed. ........................................ 56
List of Figures

2.1 Developer’s target context (paste seed: `getMethodSign(...)`) .............................. 8
2.2 Original context, `org.eclipse.jdt.internal.debug.ui.actions.-` 
`BreakpointMethodLocator (copy seed: `computeMethodSignature(...)`) .... 9
3.1 The unification and anti-unification of structures $F$ and $G$. ................................. 14
3.2 The anti-unification of the structures $\Delta (F, G)$ and $\Delta (H, G)$. .................. 15
3.3 The anti-unification of the structures $\Delta (F, G)$ and $\Delta (H, F)$. .................. 15
3.4 The anti-unification of the structures $\Delta (F, G)$ and $\Delta (G, F)$. .................. 16
3.5 The anti-unification of the structures $\Delta _A (F, G)$ and $\Delta _B (F, G)$. .......... 18
3.6 The application of the equivalence equation $= _E$ over the structures $\Delta (F, G)$ and $\Delta (G, F)$. .................................................. 18
3.7 The anti-unification of the structures $\Delta _{\text{shrink}}(\text{NIL}, I)$ and $\Delta (F, I)$. 19
3.8 The anti-unification of the structures $\Delta (K, V_a, H, \Delta _{\text{shrink}}(\text{NIL}, I))$ and $\Delta (G, V_b, H, \Delta (F, I))$. ........................................ 20
3.9 The anti-unification of the structures $\Delta _f(\Delta _g(F, G), \Delta _g(I, J))$ and $\Delta _f(\Delta _g(F, J), \Delta _g(K, L))$ that produces multiple MSAs. ................................................. 20
3.10 Simple AST structure. .......................................................... 22
3.11 The anti-unification of the structures $\Delta _C(F_A, F_k, \Delta _M(P_0, P_1, \Delta _B(S_0, \Delta _f(S_1 = P_0, S_2, S_3))))$ and $\Delta _C(F_B, F_k, \Delta _M(P_0, \Delta _B(\Delta _{\text{shrink}}(\text{NIL}, \Delta _{\text{shrink}}(\text{NIL}, \Delta _{\text{shrink}}(\text{NIL}, S_3))))))$. 23
3.12 The anti-unification of the structures $\Delta _C(\Delta _f(A, E, G), \Delta _M(\Delta _f(H, B), \Delta _B(\Delta _{\text{vd}}(I, J, D))))$ and $\Delta _C(\Delta _M(\Delta _{\text{vd}}(A, B, D)))$ that produces multiple MSAs. .................... 25
4.1 Overview of the correspondence connections that get created during the anti-
unification process to determine the structural correspondence of two ASTs. ... 28
4.2 The ASTs of the “copy & paste” seed method parameters `MethodDeclaration`
method and `MethodDeclaration node` from the motivational scenario (see
Chapter 2). .......................................................... 31
4.3 The anti-unification algorithms presented in Section 4.2 applied against the
method parameters of the “copy & paste” method seed (see Section 4.1). ... 33
4.4 The anti-unification of method parameter structures
$\Delta _P(T_{\text{MethodDeclaration}}, N_{\text{node}})$ and $\Delta _P(T_{\text{MethodDeclaration}}, N_{\text{method}})$ from
Section 4.1. .......................................................... 35
4.5 This `while`-loop maintains a similar contextual meaning to the `for`-loop on
Line 18 in Figure 2.2. .................................................. 37
4.6 The `for`-loop from Line 18 in Figure 2.2 abbreviated. ................................ 38
4.7 The conflict resolution view requests developer input to select a single anti-
unifier. .......................................................... 41
4.8 The `CREATE-RELEVANCE-LINK` algorithm applied to the CAST of the
method parameters of the “copy & paste” method seed (see Section 4.1). ... 43
4.9 Jigsaw’s output based on the motivational scenario of Chapter 2. .......... 46
4.10 Semantically equivalent `if`- and `switch-statements`. ........... 47
4.11 Jigsaw’s output based on the example classes from the semantically equivalent statements in the Figure 4.10. ....................................................... 48
4.12 Example of instances of Java programs where the VariableDeclaration in the Copy class anti-unification with the Field, Parameter, and VariableDeclaration from the Paste class can result in multiple MSAs. ............... 49
4.13 Jigsaw’s output based on the example classes from the multiple MSAs scenario in Figure 4.12. ................................................................. 50
5.1 Paste seed from test case 1 (org.eclipse.jdt.apt.core.internal.declaration.MethodDeclarationImpl.toString()). ..................... 54
## List of Algorithms

1. Input into \texttt{INTEGRATE} \texttt{(copy,paste)} are copy & paste ASTs seeds and contexts as tuples (i.e., \texttt{(‘seed,’context)}), the algorithm determines the desired missing functionality through anti-unification and the surrounding contextual information and integrates the missing functionality. ................................................. 29
2. \texttt{JIGSAW-ANTI-UNIFY} \texttt{(copy,paste)}, determines how anti-unification occurs based on the input seeds. ................................................................. 30
3. \texttt{ANTI-UNIFY} \texttt{(a,b)}, takes in two AST nodes and finds the proper comparator to compute the similarity. ................................................................. 32
4. \texttt{COMPARE-SIMPLE-NAMES} \texttt{(a,b)}, comparator that uses the longest common substring to determine the similarity between two names. .................. 34
5. \texttt{COMPARE-OTHER-LEAVES} \texttt{(a,b)}, comparator that determines the similarity of two leaf nodes. ................................................................. 34
6. \texttt{COMPARE-NON-LEAVES} \texttt{(a,b)}, comparator that determines how similarity is calculated between subtrees. ................................................................. 36
7. \texttt{ANTI-UNIFIER-OF-BEST-FIT} \texttt{(a)}, applies thresholding and, if needed, the developer’s input to resolve each node’s list of anti-unifiers. ................................. 39
8. \texttt{INCLUSION-THRESHOLD} \texttt{(a)}, removes anti-unifiers that are not greater than the \texttt{inclusion-threshold} value. ................................................................. 40
9. \texttt{DIFFERENCE-THRESHOLD} \texttt{(a)}, removes anti-unifiers that are not within \texttt{difference-threshold} value. ................................................................. 40
10. \texttt{CREATE-RELEVANCE-LINK} \texttt{(copy)}, creates a link between name uses and their declarations. ................................................................. 42
11. \texttt{INSERT-MISSING-NODE} \texttt{(a,b)}, determines the nodes from the reuse source that are to be copied and modified to fit into the developer’s target source. ................................. 44
Chapter 1

Introduction

In the days before a computer, the reuse of existing work (e.g., documents, manuscripts, film) was limited and physically laborious, whereby the editor would “cut” usually with scissors and apply an adhesive to “paste” the reused piece into place. This process did not facilitate modification of the reused work, thus limiting reuse to work that did not require modification to fit into a new context. For work that could be reused, it was an entirely manual effort, where the cost savings offered must have outweighed the effort to recreate the work in the new context.

Since the introduction of computers, the ability to “cut & paste” has been reduced to a couple of clicks of a computer keyboard and/or mouse. The cost associated with copy and paste is now insignificant, whereby the cost of the “copy & paste” of pages of work is equal to that of a single character. This has made the reuse of existing work easier; however, the modification of the existing work to fit into a new context still remains a manual process. To offset this cost, software developers introduce abstractions such as templates [GHJV95]. Templates provide the functionality to predefined the context in which the reuse is to occur. For example, Microsoft Word offers a wide selection of letter templates that have abstracted out the layout, which then can be applied to an existing document or used to create a new document. However, to create a new template, for instance, the frequency of reuse must be large enough to warrant the time investment and the user must also have the foresight to anticipate its future context of reuse. This is unrealistic in many cases. For example, consider an administrative assistant given the task to update an existing memo with the contents of an email conversation her boss had with a client. This task entails that the assistant integrate relevant fragments of the email into their corresponding sections of the memo. To do this, the assistant begins copying
and pasting sections of the email into the memo. For each copy and paste, the assistant must modify the context in which the email was written to that of the memo to be fully integrated. The types of modifications she performs consist of both those that are conceptually straight forward and those that are conceptually complex. The straight forward changes, such as grammatical modifications, can distract from the conceptually complex questions, such as “Do these changes make sense with respect to the surrounding context?” or “Do the new changes give the memo the intended meaning and purpose?”

Software developers look to reuse existing source code to reduce time and effort, and to achieve enhanced quality, reduced maintenance effort, and lower error rates [BP89, Kru92]. Software developers have developed many tools (e.g., frameworks, libraries, refactorings) to help fellow developers harness existing source code. As in the template example above, these tools also suffer some failings, as it is impossible to envision all the possible ways a piece of functionality can be reused [HW07]. For the situations where these tools either do not exist or are not applicable, developers turn to “copy & paste”. Developers have been reported to use “copy & paste” to perform on average 4 non-trivial reuses per hour [KBLN04]. “Copy & paste” allows developers to harness all the benefits of reusing existing source code; however, manual effort is required by the developer to integrate the reused source into the target context. Similar to the example of the administrative assistant, developers must perform the conceptually straight forward and the conceptually complex modifications, which have the possibility of unknowingly introducing errors [FPB87]. Currently there is no tool support to assist developers performing the integration of reused source code into the target context. Tool support could remove the developer from the conceptually straight forward changes (i.e., updating variable names) and focus their attention on the conceptually complex questions (i.e., “is this my desired result?”). This work addresses this shortcoming.
1.1 Background of small-scale reuse

Reuse is a goal that has traditionally been approached through explicit abstraction and pre-planning [FK05]. Abstraction is a heavy weight process that involves identifying and generalizing the (potentially) reused source code so that it could be used in many different contexts with little or no modification. Abstractions require a considerable amount of effort to create [TBG04]. Often there exist logical constraints that make an abstraction impossible to implement [TBG04, KSNM05]. Pre-planning requires developers to have foresight into how their source code will one day be reused. To justify the cost of such engineering efforts, reuse must occur on a large-scale in general: either large, reusable components must be provided, or smaller components must be reused frequently [Big94]. However, in practice, developers also need to reuse smaller pieces of functionality, on the scale of individual methods and classes [HW07]. Such instances of small-scale reuse cannot always justify the cost in engineering reusable components, and may not even be amenable to refactoring into reusable components [KSNM05, KG06].

Small-scale reuse tasks are performed by developers regularly during their development activities [KBLN04, HW07]. For example, understanding how to interact with an application programming interface (API) often begins with finding examples of its use [RC96, HWM06]. Once a useful example has been found, the steps involved in modifying it to integrate within the target system consist of both those that are conceptually straightforward (e.g., updating references) and those that are more complex (e.g., “is this subset of the calls really necessary in my context?”). Even conceptually straightforward modifications can cause problems: the detailed connections between the developer’s target source and the original source may be missed by the developer who is focused on the more conceptually complex issues. In paying less attention to the simple changes, they are more likely to make a mistake in doing them; however, simply spending more time on the simple changes would result in having less time to devote to careful consideration of the complex ones.
1.1.1 Overview of related work

Reuse tasks can be divided into two phases: location and integration of source code. The location phase refers to the identification of source code (e.g., frameworks, patterns, packages, classes, methods, code fragments) to be reused. The integration phase is the modification of the reused source to fit in the desired context. Much previous research has focused on the location phase (e.g., [BCC92, Mic01, HWM06, XP06]); relatively little work has considered the integration phase. Notable exceptions are approaches requiring formal specifications (e.g., [GH91, YS97]); in our context, such formal specifications cannot be expected to exist a priori and their cost to develop a posteriori would likely outweigh the benefit from the tool support. Other approaches allow developers to plan and enact medium- to large-scale reuse tasks at the integration phase (e.g., [HW07, HW08]), which is likely too heavyweight for small-scale tasks.

1.2 Broad thesis overview

Small-scale reuse generally involves modifying source code to integrate it within a target system. To perform these modifications, we can harness knowledge about the syntax and semantics of a language to infer the needed changes. To investigate how the syntax and semantics of a programming language can be used, we looked to our previous research [CCWD07] that considered how two classes can be compared to determine their detailed structural correspondences through the use of approximated anti-unification for generalization tasks. Detailed structural correspondence can be used to generalize the original and target context such that the desired functionality a developer wishes to reuse would have no correspondence. However, the problem of small-scale reuse is more complex, as integration is influenced by the surrounding context of the original system and of the target system. Therefore, an approach using detailed structural correspondence must also take into account the surrounding context.
Our approach to small-scale reuse proceeds in three steps. First, it determines candidate correspondences between structures (e.g., expressions, statements, declarations) in the original and the target systems, using a measure of similarity that considers both structural information about the structures and how they are used within their context. Structures that have a similarity value above a threshold are considered correspondence candidates. Second, when a particular structure is a candidate to correspond with more than one other structure, a conflict exists that must be resolved; the developer is presented with details of the conflict and is asked to resolve it in favour of one of the candidates. Third, the correspondences are used to identify which structures in the original system must be copied to the target system (i.e., the non-corresponding terms), and how they should be transformed to integrate with the target system. For example, references to local variables in the original system must be replaced by references to local variables in the target system.

1.3 Thesis statement

The thesis of this work is that developers can be better supported in performing reuse tasks by a technique in which structural correspondences inform a semi-automated approach for integrating source code, that focuses their attention on the most difficult issues.

1.4 Structure of thesis

The remainder of the thesis is structured as follows. Chapter 2 further motivates the problem through a small-scale source code reuse scenario. The scenario details the process that a developer follows to perform small-scale reuse; outlines potential problems that may arise; and demonstrates how the current, manual process for performing small-scale reuse is inadequately supported.
Chapter 3 provides a formal definition of anti-unification and shows why its limitations with respects to semantics of the structures in our problem context (i.e., source code) cause it to be insufficient to address our problem. We then extend our definition of anti-unification to higher-order anti-unification modulo theories and show how this extension provides a robust theoretical framework that can be applied to work with the structures (i.e., source code) in our problem context.

Chapter 4 describes our implementation of this approach in a tool, called Jigsaw, deployed as a plug-in to the Eclipse integrated development environment (IDE). Jigsaw uses a “copy & paste” metaphor, whereby the developer indicates the method to be reused (the “copy” seed) and either the class where the method should be integrated or a specific method that will act as the skeleton on which the copy will be integrated (the “paste” seed). Jigsaw then allows the developer to focus his attention on the high-level differences (e.g., inheritance hierarchies, control flow idioms, exception handling details) between the two seeds, while automatically integrating the rest.

Chapter 5 presents two empirical studies conducted to evaluate our approach. The first study evaluated the tool’s ability to perform small-scale reuse and to reproduce the results of suspected cases of small-scale reuse in an industrial system. The second involved case studies with two industrial developers who performed small-scale reuse tasks (a selection of the cases found in the first study) through the use of our tool.

Chapter 6 details related work and how it does not adequately address the problem. Chapter 7 discusses remaining issues and future work.
Chapter 2

Motivational Scenario

Software is increasingly important in the world around us. As a result, the pressure for software to do more and to provide more features is ever increasing. Each feature requires additional instructions to be written in the form of source code. Source code is expensive to create and to test, so while each new feature is necessary to meet demand, it is also risky: errors can be introduced, and the cost of development can be badly misjudged. Software reuse is a traditional idea for reducing these risks, wherein existing, well-tested source code is located and reused in a new context [JC92]. Software reuse can save the developer time and increase the quality of their software system [BP89, Kru92].

The Eclipse Java Development Tools\(^1\) (JDT) framework provides a comprehensive toolset for the creation or manipulation of source code written in the Java programming language. The JDT’s abstract syntax tree (AST) represents the specifics of source code written in the Java programming language as a tree structure. The AST offers many benefits to developers working directly with the source code: they do not need to write a language parser; it provides bindings between name and type references; and it provides mechanisms to traverse and modify nodes in the tree.

Consider a developer trying to generate a Java method signature from an AST using the JDT without tool support for small-scale reuse.

Consulting the JDT’s API he learns that the JDT uses the Visitor design pattern, provided via the ASTVisitor class. Extending the ASTVisitor he overrides a visit(...) method for capturing occurrences of the MethodDeclaration node. He decides to pass the MethodDeclaration to a getMethodSign(...) method to extract the string representation

\(^{1}\)http://www.eclipse.org/jdt, last accessed on 2008/08/06
public class MethodSignVisitor extends ASTVisitor{
  
  LinkedList<String> strList;
  
  public MethodSignVisitor(){
    strList = new LinkedList<String>();
  }
  
  public boolean visit(MethodDeclaration node) {
    strList.add( this.getMethodSign(node) );
    return super.visit( node );
  }
  
  public String getMethodSign(MethodDeclaration method){
    StringBuffer signature= new StringBuffer();
    signature.append( getParent(method) );
    
    List parameters = method.parameters();
    for (Iterator iter = parameters.iterator(); iter.hasNext();)
    {
      // TODO: Complete this body
    }
    
    return signature.toString();
  }
}

Figure 2.1: Developer’s target context (paste seed: getMethodSign(...))

of the method signature. He has started to sketch out the method when he becomes stuck, unsure how to access the information contained in the MethodDeclaration’s parameters list through a for-loop (Figure 2.1, Lines 18–20).

The API documentation does not provide any immediate guidance, so he turns to Strathcona [HWM06], a context-sensitive example recommendation tool. Using the getMethodSign(...) method as the query, Strathcona returns 10 recommended examples. He identifies the second recommended example (from the computeMethodSignature(...) method in the org.eclipse.jdt.internal.debug.ui.actions.BreakpointMethodLocator class) as potentially useful, and proceeds to examine its source code in detail (shown in Figure 2.2). From this method he identifies how to access the parameter information through a for-loop similar to the one he laid out, but he is unsure if it provides the functionality he wants until it is
```java
public class BreakpointMethodLocator extends ASTVisitor {

    private String computeMethodSignature(MethodDeclaration node) {
        if (node.getExtraDimensions() != 0 ||
            Modifier.isAbstract(node.getModifiers())) {
            return null;
        }
        StringBuffer signature = new StringBuffer();
        signature.append('(');
        List parameters = node.parameters();
        for (Iterator iter = parameters.iterator(); iter.hasNext();)
            Type type = ((SingleVariableDeclaration) iter.next()).getType();
            if (type instanceof PrimitiveType) {
                appendTypeLetter(signature, (PrimitiveType) type);
            } else {
                return null;
            }
        signature.append(')');
        Type returnType;
        returnType = node.getReturnType2();
        if (returnType instanceof PrimitiveType) {
            appendTypeLetter(signature, (PrimitiveType) returnType);
        } else {
            return null;
        }
        return signature.toString();
    }

    private void appendTypeLetter(StringBuffer signature, PrimitiveType type) {
        ...
    }

    private String computeMethodSignature2(MethodDeclaration node) {
        ...
    }

    private void appendTypeLetter2(StringBuffer signature, PrimitiveType type) {
        ...
    }
}

Figure 2.2: Original context, org.eclipse.jdt.internal.debug.ui.actions.BreakpointMethodLocator (copy seed: computeMethodSignature(...))
```
integrated within his own code. The developer has high confidence in the quality of this code fragment because it was created by the JDT developers themselves.

The developer considers refactoring the `computeMethodSignature(...)` method via “pull-up method” [Fow99, Chapter 11], but this presents many hurdles: the code is contained in an Eclipse package outside the developer’s control and the API has been marked as internal; the method was written for a different problem context; a large investment would be required to reuse the small code fragment; the developer’s use is likely to quickly diverge. These hurdles make traditional refactoring techniques unreasonable for the developer in his attempt to reuse this small snippet of code.

Instead, the developer decides to integrate `BreakpointMethodLocator.computeSignature(...)` into his own code (using `getMethodSign(...)` as a skeleton) by going through `computeMethodSignature(...)` line-by-line. He first starts by comparing the method signatures. He sees that the methods have a corresponding return type and that the parameters are nearly identical, differing only by the variable name. The developer has to remember to replace all the references to the variable `node` with ones to `method` if and when he decides to reuse the code in his own system.

Proceeding into the method body he comes across an `if`-statement on Line 4 in Figure 2.2 that does not correspond to anything in his own method; looking a little further ahead he notices that the `StringBuffer` variable declaration statement on Line 8 in Figure 2.2 is identical to his own `StringBuffer` variable declaration statement on Line 14 in Figure 2.1. With this information he decides to copy & paste the `if`-statement above his `StringBuffer` variable declaration statement, replacing the two uses of the variable `node` with uses of the variable `method`.

Next, he investigates the `signature.append(...)` method call on Line 9 in Figure 2.2, comparing it with his own usage of `signature.append(...)` on Line 15 in Figure 2.1; he determines that these method calls differ sufficiently to indicate different intents, and so he
decides to include the method call into his target source. He notices that the following List variable declaration statement on Line 10 in Figure 2.2 corresponds with his List variable declaration statement on Line 17 in Figure 2.1. He integrates the method call `signature.append(')`) between his own method call to `signature.append(...)` and the List declaration statement.

He then realizes that the for-loop expression on Line 11 in Figure 2.2 corresponds with his own for-loop expression on Line 18 in Figure 2.1. Proceeding into the body of the for-loop he decides to integrate the enclosed statements into his own for-loop and checks for any uses of the node local variable. He now has to fix an unresolved method call to `appendTypeLetter(...)`; finding that this method in BreakpointMethodLocator has no corresponding method in his own code, he decides to reuse the entire `appendTypeLetter(...)` method.

He now compares the statement following the for-loop with his own and finds that only the return statement on Line 27 in Figure 2.2 corresponds with the return statement on Line 22 in Figure 2.1. Statements from Line 19 to Line 26 in Figure 2.2 have no correspondence, so he decides to reuse the statements. He goes through the new statements, replacing uses of node with uses of method; he also identifies another use of the method `appendTypeLetter(...)`, giving him more confidence in his earlier decision to reuse that method.

The developer now realizes that the solution is partial, only providing support for primitive types; however, the result identifies where the key gaps remain to enact the desired solution. He is now left with two options: he can use this as a new starting point and fill in the gaps to produce a complete solution, or he is forced to go back over his method remembering all the changes that he has made to revert the method back to its original form.

This scenario consists of a large collection of tiny decisions and actions. Most of them are trivial, the main exception being the need to realize that the calls to `signature.append(...)` differ conceptually. However, their quantity and apparent triviality cause them to be tedious and error-prone. A better option is needed than to manually perform such drudgery.
2.1 Summary

This motivational scenario highlights the heavy burden the integration of source code places upon developers, drawing their attention away from the conceptually complex questions towards the menial tasks of the integration of the reused source code.
Chapter 3

Theoretical foundations

Programming languages, in particular Java, are described through the use of syntax which can be abstracted into structures that contain substructures, variables, and constants; therefore, a program represents an instance of a structure. To use structures as a way to help solve our problem of small-scale reuse we must address these questions: “What common elements do structures share?”, “What does it mean for two structures to be equivalent?”, “How do we determine if two structures are equivalent?”, and “Can we abstract out common structures?”.

3.1 Structure

To answer the question “What common elements do structures share?”, we first introduce a formal definition of structure, as a set of basic structures $\mathcal{B}$ (which we will refer to as base elements) and a set of combination operators $\mathcal{C}$, such that the operators come with cardinality or an unlimited number of arguments. Therefore, if we are given some structures $F$, $G$, $H$ and the combination operator $\Delta$ (which allows for an unlimited number of arguments), we can create new structures $\Delta(F,G,H)$, $\Delta(F,\Delta(G,H))$, $\Delta(H,\Delta(F,G))$, $\Delta(F,\Delta(G,\Delta(H)))$, etc. These structures all share common elements and the combination operator; where they differ lies in their construction, which leads to the question: “How do we determine if two structures are equivalent?”; or, for example, “Is $\Delta(F,G,H)$ equivalent to $\Delta(F,\Delta(G,H))$?”.

3.2 Anti-unification

To answer the questions regarding equivalence of structures we look into the theory of unification. Unification utilizes variables to perform substitutions; therefore, we must first expand our
definition of a structure. An extended structure is made up of the base elements \( B \), combination operators \( C \), and variables \( V' \). With the inclusion of variables \( V' \) in our structure, we can now define substitutions \( \sigma \) such that a variable \( X \) can be substituted by any structure, for example \( \sigma = \{ X \leftarrow F \} \).

If we unify two structures \( F \) and \( G \) then there has to exist an instantiation \( H \) where \( H = \sigma_1(F) = \sigma_2(G) \) (see Figure 3.1). Usually unification aims to create the most general unifier (MGU); that is, \( H \) is MGU (of the two structures), if \( H' = \sigma(H) \) for all unifiers \( H' \) of the structures (with appropriate substitutions \( \sigma \)). The MGU is interesting for many applications; it is however not helpful in our problem context as we are trying to create a more abstract structure (the inverse of unification) \( A \) that contains the pieces “common” to two structures \( F \) and \( G \). The inverse of unification is called anti-unification, defined so that both original structures are instances of this new structure \( A \) : \( F = \sigma'_1(A) \) and \( G = \sigma'_2(A) \) (see Figure 3.1).

![Figure 3.1: The unification and anti-unification of structures \( F \) and \( G \).](image)

For example, given two structures \( \Delta(H,G) \) and \( \Delta(F,G) \), anti-unification will create as the third structure \( A = \Delta(V,G) \), containing the common pieces of the two original structures (\( \Delta \) and \( G \)) a new variable \( V \) that can be substituted by the structures \( F \) and \( H \), to re-create \( \Delta(H,G) \) (with \( \sigma_1 = \{ V \leftarrow H \} \)), respectively \( \Delta(F,G) \) (with \( \sigma_2 = \{ V \leftarrow F \} \)) as described in Figure 3.2.
This means that the new structure $\Delta(V,G)$ (also known as an “anti-unifier”) contains the pieces of the original structures that they have in common, while abstracting from the pieces where the structures do not agree using new variables. Note that the new variables establish connections between pieces in one of the original structures and pieces in the other structure (which we will make use of later).

Because we can substitute variables for structures, we can always abstract structures by creating a variable $X$, where $X$ represents all possible anti-unifiers (shown in Figure 3.3). Therefore, similar to unification, we are interested in finding the inverse of the MGU called the most specific anti-unifier (MSA) $A$, such that for all anti-unifiers $A'$ there is a substitution $\sigma$, such that $A = \sigma(A')$.

Figure 3.2: The anti-unification of the structures $\Delta(F,G)$ and $\Delta(H,G)$.

Figure 3.3: The anti-unification of the structures $\Delta(F,G)$ and $\Delta(H,G)$.
Anti-unification provides us with a mechanism for measuring structural similarity as it provides a concrete way two structures can be made equal [Wag02]. We can use the number of common pieces over the total number of pieces of the anti-unifier of two structures to produce a measure of similarity. However, this measure alone is often not enough, as we are interested in creating a similarity measure that incorporates all knowledge we have about the structures, and anti-unification does not address concepts such as semantics of structures. For example, let us look at the structures $\Delta(F,G)$ and $\Delta(G,F)$ where the ordering of the arguments of the combination operator is insignificant from a semantic point of view. Because we do not have an equivalence for different orderings, the anti-unification of the two structures results in $\Delta(V_1,V_2)$, so $F$ and $G$ are both abstracted into variables (see Figure 3.4).

$$\Delta(V_1,V_2)$$

$\sigma_1 = \{V_1 \leftarrow F, V_2 \leftarrow G\}$

$\sigma_2 = \{V_1 \leftarrow G, V_2 \leftarrow F\}$

$\Delta(F,G)$ instance

$\Delta(G,F)$ instance

Figure 3.4: The anti-unification of the structures $\Delta(F,G)$ and $\Delta(G,F)$.

In our problem context of small-scale reuse we are dealing with two structures of varying size and complexity, an original and a target structure. The goal is to abstract the common pieces from the two structures and to apply the differences from the original structure into the target structure by means of integration. Anti-unification is a step in the right direction, as we can abstract “common pieces” in the set of structures, but is not expressive enough to handle all the knowledge we have about structures [CCWD07]. For example, the order of arguments (described above), different combination operators (e.g., Let us look at the structures $\Delta_{\text{while}}(F)$ and $\Delta_{\text{for}}(F)$ that represent two kinds of looping structures in our problem context, “Is $\Delta_{\text{while}}(F)$ equivalent to $\Delta_{\text{for}}(F)$?”), the nesting of structures (e.g., Given the
structures \(\Delta(K, \Delta(J, \Delta(H, I), F)) \) and \(\Delta(G, \Delta(H, I, F))\), “Can we create substitutions so that the common pieces can be abstracted?”), and missing structures (e.g., “Is \(\Delta(H, \Delta(F))\) equivalent to \(\Delta(H, \Delta(G, \Delta(F)))\)?”) that are found in our problem context. In the following sections we will look at extending anti-unification to higher-order anti-unification and in addition to the creation and handling of theories and how application of higher-order anti-unification modulo theories can be applied to our problem context.

3.3 Higher-order anti-unification modulo theories

The question “How do we determine if two structures are equivalent?”, still remains unanswered, as anti-unification has no mechanism to deal with concepts such as semantic knowledge. To deal with these concepts we will apply higher-order anti-unification modulo theories (also called E-generalization) [Plo70].

With anti-unification we could only substitute structures and substructures with variables \(\mathcal{V}\) limiting the types of abstractions we could perform. The structures that can be created from programming languages, in our context specifically Java, employ a vast array of combination operators. These combination operators usually come with some semantic knowledge, for example, the ordering of the structure’s arguments (discussed in Section 3.2). With higher-order anti-unification we can extend substitutions to include combination operators. To perform such substitutions we first must extend our definition of a structure to include a set \(C\mathcal{V}\) of variables for combination operators from \(C\), such that variable \(X_C\) can be substituted by any combination operator, for example \(\sigma = \{X_C \leftarrow \Delta\}\). More precisely, given the structures \(\Delta_A(F, G)\) and \(\Delta_B(F, G)\) the higher-order anti-unifier is \(\Delta_{\mathcal{V}}(F, G)\), to re-create \(\Delta_A(F, G)\) (with \(\sigma_1 = \{\Delta_{\mathcal{V}} \leftarrow \Delta_A\}\)), respectively \(\Delta_B(F, G)\) (with \(\sigma_2 = \{\Delta_{\mathcal{V}} \leftarrow \Delta_B\}\)) as shown in Figure 3.5.

Therefore we can now define our higher-order extended structure to be the base elements \(\mathcal{B}\), combination operators \(\mathcal{C}\), variables \(\mathcal{V}\), and combination variables \(C\mathcal{V}\).
The addition of the substitutions for combination operators is still not enough; for example we still cannot answer the question “Is $\Delta(F,G)$ equivalent to $\Delta(G,F)$?” (with yes in the case that the argument ordering does not matter for $\Delta$). To utilize the semantic knowledge we have about structures, we turn to anti-unification modulo theories. We describe these theories as a set of equations $E$ that can be applied over our higher-order extended structure. We can determine the equivalence of the structures through the creation of equivalence equations $=_{E}$ from above, for example $\Delta(X,Y) =_{E} \Delta(Y,X)$ to express that the ordering of the arguments is commutative (i.e., does not matter). More precisely, looking at the structures $\Delta(F,G)$ and $\Delta(G,F)$ and the equivalence theory $=_{E}$, we can create a substitution $A$ that for all $A'$, $A' = \Delta(F,G)$ or $A' = \Delta(G,F)$ (described in Figure 3.6).

![Figure 3.5: The anti-unification of the structures $\Delta_A(F,G)$ and $\Delta_B(F,G)$.](image)

We can now answer the question of equivalency between structures with different argument orderings, so now we look to structures that have common substructures but where
the non-similar pieces are very different in size, for example the structures $\Delta(K, V_a, H, I)$ and $\Delta(G, V_b, H, \Delta(F, I))$ where the substructure $I$ is common in both.

With our new concepts we can deal with these structures that have common substructures but not where the non-similar parts are very different in size. This is done by introducing a theory that adds the concept of a NIL structure for a particular combination operator $\Delta_{\text{shrink}}$. The NIL structure with its operator is used to represent the nesting and ordering of structures that exist in one structure but not the other. Looking at our previous example with the structures $\Delta(K, V_a, H, I)$ and $\Delta(G, V_b, H, \Delta(F, I))$, we will begin by focusing on the structure $I$. To create an equivalent structure to $I$ using the NIL structure we also need to introduce an equivalence equation $\sigma_1 = \{\Delta_C \leftarrow \Delta_{\text{shrink}}, X \leftarrow \text{NIL}\}$ for NIL and its particular operator $\Delta_{\text{shrink}}$. Therefore, using the NIL structure, the combination operator $\Delta_{\text{shrink}}$ and equivalence equation $\sigma_1$, we can create the substitution $\Delta_{\text{shrink}}(\text{NIL}, I)$, where we have the equality $\Delta_{\text{shrink}}(\text{NIL}, I) =_E I$. Going back to our two example structures, looking at the substructures $I$ and $\Delta(F, I)$, by substituting $I$ with $\Delta_{\text{shrink}}(\text{NIL}, I)$ we can create the anti-unifier $\Delta_C(X, I)$ (described in Figure 3.7).

$$\sigma_1 = \{\Delta_C \leftarrow \Delta_{\text{shrink}}, X \leftarrow \text{NIL}\} \quad \sigma_2 = \{\Delta_C \leftarrow \Delta, X \leftarrow F\}$$

$$\Delta_{\text{shrink}}(\text{NIL}, I) \quad \Delta(F, I)$$

Figure 3.7: The anti-unification of the structures $\Delta_{\text{shrink}}(\text{NIL}, I)$ and $\Delta(F, I)$.

Let us look back again at the original structures in the example $\Delta(K, V_a, H, I)$ and $\Delta(G, V_b, H, \Delta(F, I))$ to create the anti-unified structure. We begin by once again substituting $I$ with its identity $\Delta_{\text{shrink}}(\text{NIL}, I)$, such that our structure now becomes $\Delta(K, V_a, H, \Delta_{\text{shrink}}(\text{NIL}, I))$. Anti-unifying the structures $\Delta(K, V_a, H, \Delta_{\text{shrink}}(\text{NIL}, I))$ and $\Delta(G, V_b, H, \Delta(F, I))$, we create the anti-unifier $\Delta(V_1, V_2, H, \Delta_C(X, I))$ (described in Figure 3.8).
Higher-order anti-unification modulo theories provides a robust theoretical framework for abstracting two structures, such that we can abstract any structure using variable substitutions. However, with the introduction of combination variables $CV$ through the extension to higher-order and the equational theories, the added combination variables also add to the complexity of anti-unification of two structures by having multiple MSAs. For example let’s look at the structures $\Delta_f(\Delta_g(F,G),\Delta_g(I,J))$ and $\Delta_f(\Delta_g(F,J),\Delta_g(K,L))$ where $\Delta_f$ is commutative. We can anti-unify $\Delta_g(F,J)$ with $\Delta_g(F,G)$ to get the anti-unifier $\Delta_g(F,X)$ and then anti-unify $\Delta_g(K,L)$ with $\Delta_g(I,J)$ to get the anti-unifier $\Delta_g(Y,Z)$; or we can anti-unify $\Delta_g(F,J)$ with $\Delta_g(I,J)$ to get the anti-unifier $\Delta_g(X,J)$ and then anti-unify $\Delta_g(K,L)$ with $\Delta_g(F,G)$ to get the anti-unifier $\Delta_g(Y,Z)$ described in Figure 3.9. Having multiple MSAs adds the question “Which anti-unifier, $\Delta_f(\Delta_g(F,X),\Delta_g(Y,Z))$ or $\Delta_f(\Delta_g(X,J),\Delta_g(Y,Z))$ is most appropriate?”.

Figure 3.8: The anti-unification of the structures $\Delta(K,V_a,H,\Delta_{shrink}(\text{NIL}, I))$ and $\Delta(G,V_b,H,\Delta(F,I))$.

Figure 3.9: The anti-unification of the structures $\Delta_f(\Delta_g(F,G),\Delta_g(I,J))$ and $\Delta_f(\Delta_g(F,J),\Delta_g(K,L))$ that produces multiple MSAs.
We are interested in finding a single MSA that is most helpful for our problem context of small-scale reuse; however, the complexity of determining an MSA is undecidable in general [Bur05]. Therefore, to find an MSA that is most helpful we must fuse the higher-order extended structure with our problem structures, and use approximation techniques that are specific to our problem context and these structures.

3.4 Abstract syntax tree

Source code is represented in a tree-based intermediate representation called an abstract syntax tree. Similar to our recursive definition of structures in Section 3.2, an AST structure is made up of nodes that can be either a tree or a leaf structure. However, this basic definition of a tree is not enough for us to harness higher-order anti-unification, thus our tree definition must be extended similarly to the higher-order extended structure. A higher-order AST is a tree containing our base elements $\mathcal{B}$ (e.g., fields, parameters, statements), substitution variables $\mathcal{V}$, combination operators $\mathcal{C}$ (e.g., blocks, class, methods), and combination operator variables $\mathcal{CV}$, such that an instance of an AST for the Java programming language can be represented in this abstract form:

$$\Delta_C(F_1, \ldots, F_n, \Delta_M(P_{M_1,1}, \ldots, P_{M_1,k}, \Delta_B(S_{M_1,1}, \ldots, S_{M_1,l})), \ldots, \Delta_M(P_{M_m,1}, \ldots, P_{M_m,i}, \Delta_B(S_{M_m,1}, \ldots, S_{M_m,j})))$$

where,

$F$ represents fields

$S$ represents statements

$P$ represents parameters

$\Delta_M$ represents methods

$\Delta_C$ represents a class

$\Delta_B$ represents a block
For example, we can represent the simple AST structure described in Figure 3.10 as

\[ \Delta_C(f, \Delta_M(\text{arg}, \Delta_B(S_0, \Delta_{if}(S_1 = \text{arg}, S_2, S_3), \Delta_{loop}(S_4, S_5), S_6))). \]

Figure 3.10: Simple AST structure.

The Java programming language provides many different ways that loops, control, language specific variables, expression, and name structures can be defined while maintaining the same semantics. To deal with these semantics of the Java language we also need to include our set of theories \( \mathcal{E} \) that can be applied over our higher-order AST structure. Similar to the creation of the equivalence equations \( =_\mathcal{E} \) in Section 3.3, we will also need to add to this set to resolve different structures that maintain the same semantic meaning. For example, let
us look at the if- and switch-structures that we find in our AST structure. We have the
structures \( C, B_1, \) and \( B_2, \) where \( C \) is the branching condition, \( B_1 \) is the branch that is executed
if the condition is true, and \( B_2 \) is the branch that is executed if the condition is false. We
can create an if-structure \( \Delta_{if}(C, B_1, B_2) \) that is semantically equivalent to a switch-structure
\( \Delta_{switch}(\Delta_{case_1}(C, B_1), \Delta_{case_2}(\neg C, B_2)) \). Therefore through the equivalence equations \( \equiv_E \), we can
create simple substitutions that can represent the semantically similar structures.

Similar to the inclusion of equivalence equations, we need to also include the
NIL-theory to deal with AST structures that have common substructures but where
the non-similar parts are very different in size. We will expand the NIL-theory
to the combination operators for our AST structures with varying numbers of arguments. For example, let us look at the structures \( \Delta_C(\Delta_{if}(S_0, \Delta_{if}(S_1 = P_0, S_2, S_3))) \) and \( \Delta_C(\Delta_{if}(S_0, \Delta_{if}(S_1 = P_0, S_2, S_3))) \). Using the NIL-theory we can create the
structure \( \Delta_C(\Delta_{if}(S_0, \Delta_{if}(S_1 = P_0, S_2, S_3))) \) that is E-equal to \( \Delta_C(\Delta_{if}(S_0, \Delta_{if}(S_1 = P_0, S_2, S_3))) \). We can also use equivalence equations \( \equiv_E \) to elimi-
nate the need to rewrite the combination operators to include the NIL-structures for
our structures with varying number of arguments, so that our structure has the form
\( \Delta_C(\Delta_{if}(S_0, \Delta_{if}(S_1 = P_0, S_2, S_3))) \). With the substitute structure we can
create the anti-unifier, \( \Delta_C(V_0, F_k, \Delta_M(P_0, V_1, \Delta_B(V_2, \Delta_Y(X, Y, S_3)))) \) described in Figure 3.11.

\[
\Delta_C(V_0, F_k, \Delta_M(P_0, V_1, \Delta_B(V_2, \Delta_Y(X, Y, S_3)))
\]

\[
\sigma_1 = \{ V_0 \leftarrow F_A, V_1 \leftarrow P_1, V_2 \leftarrow S_0, \\
\Delta_Y \leftarrow \Delta_{if}, X \leftarrow S_1 = P_0, Y \leftarrow S_2 \}
\]

\[
\Delta_C(F_A, F_k, \Delta_M(P_0, V_1, \Delta_B(\Delta_Y(X, Y, S_3)))
\]

\[
\sigma_2 = \{ V_0 \leftarrow F_B, V_1 \leftarrow \text{NIL}_{\Delta_M}, \\
V_2 \leftarrow \text{NIL}_{\Delta_B}, \Delta_Y \leftarrow \Delta_{shrink}, \\
X \leftarrow \text{NIL}_{\Delta_{shrink}}, Y \leftarrow \text{NIL}_{\Delta_{shrink}} \}
\]

\[
\Delta_C(F_B, F_k, \Delta_M(P_0, \Delta_B(\Delta_{shrink}(\text{NIL}_{\Delta_{shrink}}, \text{NIL}_{\Delta_{shrink}}, S_3)))
\]

\[
\Delta_C(F_B, F_k, \Delta_M(P_0, \Delta_B(\Delta_{shrink}(\text{NIL}_{\Delta_{shrink}}, \text{NIL}_{\Delta_{shrink}}, S_3)))
\]

Figure 3.11: The anti-unification of the structures \( \Delta_C(F_A, F_k, \Delta_M(P_0, P_1, \Delta_B(S_0, \Delta_{if}(S_1 = P_0, S_2, S_3))) \) and \( \Delta_C(F_B, F_k, \Delta_M(P_0, \Delta_B(\Delta_{shrink}(\text{NIL}_{\Delta_{shrink}}, \text{NIL}_{\Delta_{shrink}}, S_3))) \).
Small-scale reuse involves filling “holes” with the missing functionality. The problem is in determining what the “holes” are between two AST structures and how to integrate that functionality filling the “holes”. Our higher-order AST structure and set of theories $E$ allows us to abstract the similar parts of our AST structure. The application of the $\text{NIL}$-theory provides us with the location of the missing structures (i.e., the “holes” we want to fill). The anti-unifier through the use of the variables $\mathcal{V}$ and combination operator variables $\mathcal{CV}$ provides us via their substitutions with the connections needed to make the two structures equal (i.e., the structures needed to fill the “holes”). The examples above show how quickly these anti-unifiers become complicated. With each variable having an unbounded number of possible substitutions combined with our equivalence equations and $\text{NIL}$-theory, finding the anti-unifier that represents the MSA still remains undecidable [Bur05]. For example, let us look at the semantic similarities between fields ($\Delta_f$), parameters ($\Delta_p$), and variable declaration ($\Delta_{vd}$). If we have the structures $\Delta_C(\Delta_f(A,E,G),\Delta_M(\Delta_p(H,B),\Delta_B(\Delta_{vd}(I,J,D))))$ and $\Delta_C(\Delta_M(\Delta_{vd}(A,B,D)))$ and the equivalence equation $=_E$ for $\Delta_f, \Delta_p, \Delta_{vd}$. If we anti-unify $\Delta_f(A,E,G)$ with $\Delta_{vd}(A,B,D)$ to get the anti-unifier $\Delta_{V_0}(A,V_1,V_2)$; or we anti-unify $\Delta_p(H,B)$ with $\Delta_{vd}(A,B,D)$ to get anti-unifier $\Delta_{V_0}(V_1,B,V_2)$; or we anti-unify $\Delta_{vd}(I,J,D)$ with $\Delta_{vd}(A,B,D)$ to get the anti-unifier $\Delta_{vd}(V_0,V_1,D)$ described in Figure 3.12. We have three MSAs that have in “common” separate parts (i.e., type, name, and assignment) that brings us back to the question, “Which anti-unifier is most appropriate?”.

To answer these questions we must look to approximation techniques to find the anti-unifier of best fit. By applying a greedy selection algorithm over the anti-unifier, we select the best anti-unifiers for each substructure in our structure thus creating our anti-unifier of best fit. In the following chapter we will discuss Jigsaw, our implementation of higher-order anti-unification modulo theories for small-scale reuse.
3.5 Summary

This chapter formally defines anti-unification, a robust theoretical framework for abstracting structures. We have shown how extending the theory of anti-unification to higher-order anti-unification modulo theories we can include variables and create theories to incorporate semantic knowledge of structures. We also showed how higher-order anti-unification modulo theories can be applied to create abstractions of the tree-based structures (i.e., ASTs) specific to our problem context.
Chapter 4

The Jigsaw Tool

In Chapter 3 we described higher-order anti-unification modulo theories, a robust theoretical framework for creating abstractions of structures. In Section 3.4 we showed how this framework can be applied to AST structures that we use to describe an instance of a program. So how does the creation of an abstraction help us integrate missing functionality contained in reused source code into the developer’s target context?

To answer the question, let’s look at an overview of our problem and how we can utilize higher-order anti-unification modulo theories to help solve the problem of small-scale reuse. For example, a developer supplies us with two method seeds and their surrounding contexts shown in Figure 4.1(a,b) as ASTs with symbols marking differences in structures. The copy method seed (Figure 4.1(c)) represents the functionality the developer wants to reuse; and the paste method seed (Figure 4.1(d)) is the location of the “hole” that the developer wishes to fill with the functionality contained in the copy method seed. We are only interested in anti-unifying similar structures (i.e., no anti-unifiers of totally different structures abstracted into a variable \( V \)); therefore, anti-unifying the two AST structures (working down the AST structures) we find: structures that are exactly the same (Figure 4.1(e)); structures that are semantically equivalent using our equivalence equations \( =_E \) (Figure 4.1(f)); anti-unified structures (Figure 4.1(g)); two structures that are unequal and semantically unequal, such that we require a variable \( X \) in our anti-unifier (Figure 4.1(k)); structures with multiple anti-unifiers (Figure 4.1(h)); using NIL-theory, to form NIL structures that represent the “hole” in the developer’s context and connections with the structures in the copy structure needed to fill the “holes” (Figure 4.1(i)). In addition to our anti-unified structure (Figure 4.1(j)), we can add links (called relevance links) between the uses and declarations of structures within our ASTs.
(Figure 4.1(r)) to create transformation rules that can be applied to the integration of the structures into the “hole”. So our anti-unifier (Figure 4.1(j)) is a set of correspondence connections between our two AST structures. Therefore, to create an anti-unifier we do not need to create a new structure; we can harness the existing structure of our ASTs and augment each node in the AST to hold a list of their connections (i.e., each connection in a node’s list represents an anti-unifier), that we call the correspondence AST (CAST).

To perform small-scale reuse via structural correspondence as described above, we implemented a tool called Jigsaw that uses approximated higher-order anti-unification modulo theories to determine the structural correspondence between two ASTs. Jigsaw is a plug-in to the Eclipse\(^1\) integrated development environment (IDE). The developer supplies two seeds as input: (1) the method containing the desired missing functionality that is to be reused (the copy seed); and (2) either a class or a method where the missing functionality is to be integrated (the paste seed). From the point of view of Jigsaw, it is supplied with four inputs: (1) the copy seed (i.e., method); (2) the copy context (i.e., class minus the copy seed); (3) the paste seed (i.e., method or class); and (4) the paste context (i.e., class minus the paste seed, if the seed is a method).

Jigsaw performs a sequence of actions on the copy & paste seeds and contexts, outlined by the algorithm INTEGRATE: (1) input into the algorithm are the ASTs for the seeds and their contexts as tuples (i.e., \((\text{seed, context})\)), created via Eclipse’s standard functionality; (2) anti-unification of the ASTs (Line 1), producing candidate correspondences between the original and the developer’s target source (see Section 4.2) that are stored in an augmented form of an AST, the CAST; (3) evaluation of candidate anti-unifiers to obtain the “best” correspondence (Line 2) (see Section 4.3); (4) creation of relevance links (Line 3) between nodes referencing identical names (see Section 4.4); and (5) transformation of the selected anti-unifier (Lines 4 and 6 ) to propose an integration solution for the developer’s system (see Section 4.5).

\(^1\)http://www.eclipse.org, v3.3
Figure 4.1: Overview of the correspondence connections that get created during the anti-unification process to determine the structural correspondence of two ASTs.
Algorithm 1 Input into INTEGRATE\((copy, paste)\) are copy & paste ASTs seeds and contexts as tuples (i.e., \((seed, context)\)), the algorithm determines the desired missing functionality through anti-unification and the surrounding contextual information and integrates the missing functionality.

\[
\begin{align*}
\text{INTEGRATE}(copy, paste) & \\
1 & (cast[copy], cast[paste]) \leftarrow \text{JIGSAW-ANTI-UNIFY}(copy, paste) \\
2 & \text{cast} \leftarrow \text{ANTI-UNIFER-OF-BEST-FIT}(\text{cast}) \\
3 & \text{CREATE-RELEVANCE-LINK}(\text{cast[copy]}) \\
4 & \text{for each } k_a \in \text{node[cast[copy]]} \\
5 & \quad \text{do if corresp}[k_a] = \{\} \\
6 & \quad \text{then INSERT-MISSING-NODE}(k_a, cast[paste])
\end{align*}
\]

To start the anti-unification process and to help limit the number of possible anti-unifiers, Jigsaw considers the nature of the seeds and their surrounding contexts, via the \text{JIGSAW-ANTI-UNIFY} algorithm. If the paste seed is a method (defined as a \text{MethodDeclaration} in the JDT AST specification), the copy & paste seed methods (\(\Delta_{m_{copy}}\) and \(\Delta_{m_{paste}}\)) are compared directly with one another (Line 2); otherwise, Jigsaw assumes there is no method in the paste class that corresponds to the copy seed and uses the \text{NIL}-theory to construct using a mirror structure \(\Delta_{\text{shrink}}\) of the copy seed \(\Delta_{m_{copy}}\). Therefore we limit the seed methods to a single anti-unifier \(\Delta_{m_{seed}}\), to re-create the \(\Delta_{m_{copy}}\) (with \(\sigma_1 = \{\Delta_{m_{seed}} \leftarrow \Delta_{m_{copy}}\}\)), respectively either \(\Delta_{m_{paste}}\) (with \(\sigma_2 = \{\Delta_{m_{seed}} \leftarrow \Delta_{m_{paste}}\}\)) for the first case or \(\Delta_{\text{shrink}}\) (with \(\sigma_2 = \{\Delta_{m_{seed}} \leftarrow \Delta_{\text{shrink}}\}\)) for the second case. Each field declaration node in the paste class AST is compared to all the field declaration nodes in the copy class AST (Lines 3–5). Each method declaration node in the paste class AST is compared with all the method declaration nodes in the copy class AST (Lines 6–8).

The anti-unification of the fields and the methods in the context class has a complexity of \(O(n^2)\) in the worst case, and if we expand it to the substructures contained with a method and then onto the nested substructures contained within those substructures and so forth our com-
Algorithm 2 JIGSAW-ANTI-UNIFY(copy, paste), determines how anti-unification occurs based on the input seeds.

\begin{algorithm}
\begin{algorithmic}[1]
\State \textbf{JIGSAW-ANTI-UNIFY}(copy, paste)
\State \textbf{if} seed[paste] instanceof MethodDeclaration
\State \textbf{then} ANTI-UNIFY(seed[copy], seed[paste])
\State \textbf{for} each $k_{copy} \in \text{fields}[copy]$
\State \quad \textbf{do} for each $k_{paste} \in \text{fields}[paste]$
\State \quad \quad \textbf{do} ANTI-UNIFY($k_{copy}, k_{paste}$)
\State \textbf{for} each $k_{copy} \in \text{methods}[copy] \setminus \{seed[copy]\}$
\State \quad \textbf{do} for each $k_{paste} \in \text{methods}[paste] \setminus \{seed[paste]\}$
\State \quad \quad \textbf{do} ANTI-UNIFY($k_{copy}, k_{paste}$)
\end{algorithmic}
\end{algorithm}

Complexity becomes exponential ($O(n^m)$). So it will be very important when creating the similarity measure to represent our equivalence equations so that the measure focuses on providing us with information useful to creating anti-unifiers that will be of interest to us.

We consider only those anti-unifiers where there exist “common” pieces between two structures to be of interest for our problem of small-scale reuse. We want to build anti-unifiers that connect the “common” pieces of our structures to identify the “holes” we are interested in filling and use that information obtained from the anti-unification to help inform and direct the integration of the missing functionality into the “holes”.

4.1 Running example

We introduce an example from the motivational scenario (see Chapter 2) to assist in explaining the anti-unification process over two ASTs. We limit the example to the method parameter subtrees from the “copy & paste” method seeds shown in Figure 4.2, “MethodDeclaration node” and “MethodDeclaration method” respectively. The method parameters are each represented as a VariableDeclaration subtree that is made up of two leaves (a Type and a Name) with their values shown in bold text.
Figure 4.2: The ASTs of the “copy & paste” seed method parameters. MethodDeclaration method and MethodDeclaration node from the motivational scenario (see Chapter 2).

The general structural representation of the ASTs as defined in Section 3.4 is \( \Delta_M(\Delta_P(\text{Type}, \text{Name}), \ldots) \), where \( \Delta_P \) represents parameters. We will define the original context structure to be \( \Delta_P(T_{\text{MethodDeclaration}}, N_{\text{node}}) \) and the developer’s target structure to be \( \Delta_P(T_{\text{MethodDeclaration}}, N_{\text{method}}) \).

### 4.2 Constructing the CAST

The goal of this phase is to determine all possible candidate correspondences between nodes in the seed ASTs; in a later phase, nodes from the copy context that are found to not correspond will be copied and integrated.

We have developed a measure of the similarity between two AST nodes, for use in the anti-unification process; the similarity measure produces values between zero and one. \textsc{Anti-Unify} computes the similarity \( \text{sim}[a, b] \) of the CAST nodes \( a \) and \( b \); if this is greater than 0, the presence of a candidate anti-unifier \( g \) between \( a \) and \( b \) is recorded in the CAST. A similarity value of 0 means that the two AST structures that were compared have no “common”
pieces (i.e., we would abstract to a variable in the anti-unifier); therefore they are not of interest to us and we do not create an anti-unifier for the two structures.

Algorithm 3 \textsc{anti-unify}(a, b), takes in two AST nodes and finds the proper comparator to compute the similarity.

\begin{algorithm}
\textsc{anti-unify}(a, b)
\begin{algorithmic}[1]
\STATE Comparator $\leftarrow$ \textsc{look-up-comparator}(\text{type}[a], \text{type}[b])
\IF {Comparator = NIL}
\STATE $g[a, b] \leftarrow$ NIL
\STATE $\text{sim}[a, b] \leftarrow$ 0
\ELSE $\uparrow$Comparator($a, b$)
\end{algorithmic}
\end{algorithm}

In general, Jigsaw compares nodes of identical type to determine how similar they are. In addition, Jigsaw uses semantically-based heuristics to permit the comparison of nodes that are not of identical type but that have related semantics (discussed in Sections 3.3 and 3.4); some of these heuristics leverage context-of-use information to improve the accuracy of the similarity measure. For example, let us supply the \textsc{anti-unify} algorithm with the method parameters from the seed methods from Section 4.1. Figure 4.3 shows how \textsc{anti-unify} is applied at each level of the AST through the use of the \textit{Comparator}.

The algorithm \textsc{look-up-comparator} selects the most appropriate comparison algorithm for nodes of type $T_1$ and $T_2$. The comparison algorithms are our concrete implementations of our equivalence equations $=_E$ that are part of our sets of theories $E$ (that we use in our anti-unification modulo theories, see Section 3.3). If none exists, the similarity of such nodes is defined to be 0. There are three basic cases: comparison of simple name nodes, comparison of other leaf nodes, and comparison of non-leaf nodes. For example, there exist comparators for \texttt{VariableDeclaration}s (i.e., \texttt{compare-non-leaves}), \texttt{Types} (i.e., \texttt{compare-other-leaves}), and \texttt{Names} (i.e., \texttt{compare-simple-name}) which are shown in Figure 4.3. Calls to the algorithm \textsc{anti-unify} that did not return a \textit{Comparator} (e.g., comparison of \texttt{Type} and \texttt{Name}) are not shown in Figure 4.3.
Figure 4.3: The anti-unification algorithms presented in Section 4.2 applied against the method parameters of the “copy & paste” method seed (see Section 4.1).

To produce a similarity measure between simple name nodes, we created an equivalence equation \( =_E \) that utilizes the largest common substring. Simple name nodes are compared on the basis of the length of their longest common substring, normalized to a value between 0 and 1, as suggested by the algorithm \textsc{Compare-Simple-Names} (Line 3). The weight \( w_N \) can be configured to any value in \([0,1]\); we have found that a value of 1 gives the best results, via informal experimentation. For example, let us supply \textsc{Compare-Simple-Names} with the \texttt{Name} nodes containing the values \texttt{node} and \texttt{method} from the ASTs in Figure 4.3, (i.e., \( N_{\text{node}} =_E N_{\text{method}} \)). \textsc{Longest-Common-Substring}(\texttt{node}, \texttt{method}) will return 2 and the \texttt{max} of the two names is 6, thus resulting in a similarity of \( \frac{1}{3} \) and the anti-unifier \( N_{V_1} \).

Other leaf nodes are compared via the \textsc{Compare-Other-Leaves} algorithm, which determines whether the simple properties of the nodes are equal (Line 1). If so, their similarity is defined as the weight \( w_L \), configured to any value \([0,1]\) (Line 2); we have found that a weight
Algorithm 4 \texttt{COMPARE-SIMPLE-\textsc{Names}(a, b)}, comparator that uses the longest common substring to determine the similarity between two names.

\textsc{Compare-Simple-\textsc{Names}(a, b)}

\begin{algorithmic}[1]
\State $s_a \leftarrow \text{string}[a]$
\State $s_b \leftarrow \text{string}[b]$
\State $\text{sim}[a,b] \leftarrow w_N \times \text{size[Longest-Common-Substring}(s_a, s_b)] / \max \{s_a, s_b\}$
\State \textbf{if} $\text{sim}[a, b] > 0$
\State \hspace{1em} $g[a,b] \leftarrow \text{Create-Anti-unifier-Annotation()}$
\State \textbf{else}
\State \hspace{1em} $\text{sim}[a,b] \leftarrow 0$
\State \hspace{1em} $g[a,b] \leftarrow \text{NIL}$
\end{algorithmic}

of 1 returns the best results, again through informal experimentation. Otherwise, their similarity is defined as 0 (Line 4). For example, let us supply \texttt{COMPARE-\textsc{Other}-\textsc{Leaves}} with the Type nodes from the ASTs in Figure 4.3 (i.e., $T_{\text{MethodDeclaration}} = T_{\text{MethodDeclaration}}$). The two types are equivalent, thus resulting in a similarity of 1 and the anti-unifier $T_{\text{MethodDeclaration}}$.

Algorithm 5 \texttt{COMPARE-\textsc{Other}-\textsc{Leaves}(a, b)}, comparator that determines the similarity of two leaf nodes.

\texttt{Compare-\textsc{Other}-\textsc{Leaves}(a, b)}

\begin{algorithmic}[1]
\State \textbf{if} $a = b$
\State \hspace{1em} \textbf{then} $\text{sim}[a, b] \leftarrow w_L$
\State \hspace{1em} $g[a,b] \leftarrow \text{Create-Anti-unifier-Annotation()}$
\State \textbf{else}
\State \hspace{1em} $\text{sim}[a, b] \leftarrow 0$
\State \hspace{1em} $g[a,b] \leftarrow \text{NIL}$
\end{algorithmic}

The comparison of non-leaf nodes proceeds recursively, via the \texttt{COMPARE-\textsc{Non}-\textsc{Leaves}} algorithm. The children of the nodes being compared are themselves pairwise compared exhaustively (Lines 2–5). For each child node of $a$, the maximum similarity to any child node of $b$ is determined (Line 6); the sets of these maxima are summed up to estimate the size of the best fit (Line 7). The similarity is then computed as the ratio of the estimated best fit possible to the perfect fit (Line 8). For example, let us supply \texttt{COMPARE-\textsc{Non}-\textsc{Leaves}} with the \texttt{VariableDeclaration} nodes from the ASTs in Figure 4.3.
(i.e., $\Delta_P(T_{\text{MethodDeclaration}}, N_{\text{node}}) =_E \Delta_P(T_{\text{MethodDeclaration}}, N_{\text{method}})$). The children of the VariableDeclarations are Type and Name respectively. The call to $\text{ANTI-UNIFY}(k_a, k_b)$ on Line 5 will return a similarity of 1 for the Type comparison and $\frac{1}{3}$ for the Name comparison (as described above). The comparison of Type to Name will result in an similarity of 0. Exiting the for-loops, the value of best is $1\frac{1}{3}$ which is then input to the calculation of the similarity on Line 8, resulting in a similarity of $\frac{2}{3}$. Thus we can represent the anti-unifier as the structure $\Delta_P(T_{\text{MethodDeclaration}}, V_1)$, described in Figure 4.4.

Figure 4.4: The anti-unification of method parameter structures $\Delta_P(T_{\text{MethodDeclaration}}, N_{\text{node}})$ and $\Delta_P(T_{\text{MethodDeclaration}}, N_{\text{method}})$ from Section 4.1.

We express our set of equivalence equations $=_E$ for comparing the combination operators $C$ through the COMPARE-NON-LEAVES algorithm. For each of the different concrete implementations of the algorithm, we apply these equivalence equations through semantically-based heuristics. These semantically-based heuristics utilize specific knowledge about our structures obtained from the Java language specification and the JDT AST specification. These specifications are used to determine the “common” pieces between two structures and then abstract out these “common” pieces for comparison. However, in our problem of reuse the developer’s target method seed contains missing information, which leads us to the question “How many common pieces do we need to exist between two structures to determine if they are semantically equivalent?”. More precisely, if we have two loop structures, a while-loop $\Delta_{\text{while}}(H, \Delta_B(\ldots))$ and a for-loop $\Delta_{\text{for}}(I, H, C, \Delta_B\ldots)$ where $I$ is the initializer, $H$ is the halting condition, and $C$
is the iterator. “Do we need the information gained from the comparisons from the bodies $\Delta_B$ of the two loops to determine if the loops are semantically equivalent?” Because we are interested in determining if the two structures have the same semantical meaning, therefore we are only interested in using the comparisons of those substructures that aid in that meaning of the parent structure. For example, looking back to our loops from above, and applying our knowledge of the Java language specification and the JDT AST specification we can abstract out the initializers, halting conditions, and iterators if they exist, to determine the anti-unification of the two loops.

From the specifications the AST nodes fall into one of these categories: variable declarations, conditionals, loops, statements, expressions, method declarations, type declarations, type references, exception handling, and import declarations.

**Algorithm 6** **COMPARE-NON-LEAVES**$(a, b)$, comparator that determines how similarity is calculated between subtrees.

```plaintext
COMPARE-NON-LEAVES$(a, b)$
1 best ← 0
2 for each $k_a \in \text{children}[a]$
3     do max ← 0
4     for each $k_b \in \text{children}[b]$
5         do ANTI-UNIFY$(k_a, k_b)$
6         max ← max$(\text{max}, \text{sim}[k_a, k_b])$
7     best ← best + max
8 $\text{sim}[a, b] \leftarrow \text{min}$$(\text{best}, |a|, |b|) / \text{max}(|a|, |b|)$
9 $\triangleright$ Each concrete implementation of COMPARE-NON-LEAVES can perform semantically-based heuristic adjustments here
10 if $\text{sim}[a, b] > 0$
11     then $g[a,b] \leftarrow \text{CREATE-ANTI-UNIFIER-ANNOTATION}()$
```

The variable declaration comparator considers all formal parameter and local variable declarations. Identity of their types is used to infer perfect similarity, while inheritance hierarchy relationships are treated as implying high similarity. For example, the formal parameter MethodDeclaration method in Figure 4.3 will also be compared with the variable declara-
tions contained within the method on Lines 8, 10, 12, and 20 in Figure 2.2. Because variable declarations have generally a smaller subtree, the possibility of having multiple MSAs increases. To select the nodes that are of more interest to us, we assume nodes that occur at the same level in the ASTs to have slightly more weight. By slightly more weight, we mean that we increment the size of the variable declaration nodes subtree by 1 and increment the value of \textit{best} by 1, if the nodes being compared occur at the same level in the COMPARE-NON-LEAVES algorithm (see Section 4.7).

The conditionals comparator allows comparison of \texttt{if} and \texttt{switch}-statements. The test condition of the \texttt{if}-statement is compared against both the test condition of the \texttt{switch}-statement and each \texttt{case}'s expression. The use of abstraction to compare the “common” pieces of these two structures is described in Section 4.6. Despite the fact that the semantic similarity of these expressions will tend to be low to non-existent, these comparisons aid to identify partial correspondences that are flagged for the developer’s attention.

\begin{verbatim}
Iterator iter = parameters.iterator();
while (iter.hasNext()) { ... }
\end{verbatim}

Figure 4.5: This \texttt{while}-loop maintains a similar contextual meaning to the \texttt{for}-loop on Line 18 in Figure 2.2.

The loop comparator allows comparison of enhanced-\texttt{for}-, \texttt{for}-, \texttt{while}- and \texttt{do}-statements. The halting conditions of these loops, in particular, are compared. The types of all references in these conditions are compared, and a simple data flow analysis heuristic is used to compare the (assumed) most recent assignments to variables that are referenced; specifically, the lexically-closest previous assignment to the variable is used for comparison. For example, consider comparing the \texttt{for}-loop Line 18 in Figure 2.2 (abbreviated form of the loop is shown in Figure 4.6) with the \texttt{while}-loop from Figure 4.5. The type \texttt{Iterator} and halting condition \texttt{iter.hasNext()} are identical. Tracing the use of \texttt{iter} in each seed, we
identify that the halting condition for each loop uses iter in an identical manner. In terms of the anti-unification process outlined in the algorithms above, the tracing of iter occurs in the \textsf{COMPARE-NON-Leaves} algorithm at \texttt{children[a]} on Line 2. The type of iter is \texttt{Iterator} in both cases that is most recently defined through a call to \texttt{parameters.iterator()}. From this information Jigsaw asserts that the while-loop provides a highly similar (i.e., similarity value of $\frac{7}{9}$) semantics and contextual use.

\begin{verbatim}
for (Iterator iter = parameters.iterator(); iter.hasNext();)
    ...
\end{verbatim}

Figure 4.6: The for-loop from Line 18 in Figure 2.2 abbreviated.

Therefore our similarity measure (which is similar to the similarity measure we defined in Section 3.2) is the number of “common” structural pieces (i.e., nodes) over the total number of structure pieces. For properties that are contained within a node or expressed about a node (e.g., the level that a node occurs) that are not expressed through the structure of the node, we abstract these properties into child structures (i.e., adding the nodes into the originating node’s subtree) to aid in the comparison of the parent structure.

Statements and expressions, not otherwise handled, are compared on the basis of the types that they resolve to; every statement is compared to every other statement, regardless of how it is nested within other nodes. Method declarations are compared solely on the basis of their signatures (i.e., their bodies do not affect their similarities). Type declarations, and type references, are compared solely on the basis of their supertype hierarchies and names.

A variety of nodes (e.g., try–catch statements) can only be compared with nodes of identical type (although their contained statements are compared with other statements, independently). However, throw and throws nodes are compared against catch nodes; our reasoning is that, if the developer’s paste seed provides a catch block for a particular exception, the throwing of that exception in the copy seed is likely inappropriate to copy into the target context as the developer is already explicitly dealing with it (the equivalent argument in the
opposite direction also holds). Import declarations are the simplest comparison of all: they are either identical or not.

4.3 Anti-unifier of best fit

A CAST constructed as described in Section 4.2 still represents many possible anti-unifiers—too many to be practical for the user. To overcome this, we create an anti-unifier of best fit via thresholding and, if needed, the developer’s input to resolve each node’s list of anti-unifiers to select the strongest. The ANTIT-UNIFIER-OF-BEST-FIT algorithm shows how the decisions made by Jigsaw or by the developer are cascaded to other anti-unifiers in the CAST to reduce the number of possible future decisions; thus, the best fit is computed greedily with respect to the similarity measure, and may not represent an optimal fit for the entire task (determining an optimal fit reduces to the bin packing problem, which is in NP-hard [Joh73]).

To reduce the number of decisions that the developer must make, we apply two thresholds over the CAST to filter anti-unifiers (Lines 2–5). Anti-unifiers in the copy class that the copy seed does not have dependencies on, will have no effect on the integration and are ignored. If the size of the set of anti-unifiers for a CAST node is still greater than one after the thresholds have been applied, the developer is asked to supply input (Lines 6 and 7).

**Algorithm 7** ANTIT-UNIFIER-OF-BEST-FIT\((a)\), applies thresholding and, if needed, the developer’s input to resolve each node’s list of anti-unifiers.

\[
\text{ANTIT-UNIFIER-OF-BEST-FIT}(a) \\
\text{1 } \textbf{for each } k \in \text{children}[a] \\
\text{2 } \textbf{do if } |\text{anti-unifiers}[k]| > 0 \\
\text{3 } \quad \textbf{then INCLUSION-THRESHOLD}(k) \\
\text{4 } \quad \textbf{if } |\text{anti-unifiers}[k]| \geq 2 \\
\text{5 } \quad \textbf{then DIFFERENCE-THRESHOLD}(k) \\
\text{6 } \quad \textbf{if } |\text{anti-unifiers}[k]| \geq 2 \\
\text{7 } \quad \textbf{then PROMPT-DEVELOPER-INPUT}(\text{anti-unifiers}[k])
\]
**Inclusion threshold.** If the similarity of an anti-unifier is below the inclusion threshold value, the anti-unifier is removed which is done by the `INCLUSION-THRESHOLD` algorithm. The algorithm traverses every candidate anti-unifier and removes the anti-unifier from the CAST (Line 3) if it is below the threshold value (Line 2). The default value for the inclusion threshold is 0.25, which means at least one-fourth of the corresponding nodes, including their children, correspond. We arrived at this value through informal calibration tests; it has performed with reasonable results in our experience.

**Algorithm 8** `INCLUSION-THRESHOLD(a)`, removes anti-unifiers that are not greater then the `inclusion-threshold` value.

```
INCLUSION-THRESHOLD(a)
1  for each k ∈ anti-unifiers[a]
2      do if sim[k] ≤ inclusion-threshold
3      then anti-unifiers[a] ← anti-unifiers[a] \ \{k\}
```

**Difference threshold.** Once the `INCLUSION-THRESHOLD` has completed traversing the CAST and if the CAST node contains more than one anti-unifier, the `DIFFERENCE-THRESHOLD` algorithm is applied. The algorithm is used to remove anti-unifiers (Line 4) if their similarity value is not sufficiently close to that of the anti-unifier (Lines 2 and 3) with the largest similarity in the list being evaluated (Line 1). The default value for the difference threshold is 0.05, which we found reasonable for most cases.

**Algorithm 9** `DIFFERENCE-THRESHOLD(a)`, removes anti-unifiers that are not within `difference-threshold` value.

```
DIFFERENCE-THRESHOLD(a)
1  best ← max(anti-unifiers[a])
2  for each k ∈ anti-unifiers[a] \ \{best\}
3      do if sim[best] − sim[k] > difference-threshold
4      then anti-unifiers[a] ← anti-unifiers[a] \ \{k\}
```
**Developer input.** If more than one anti-unifier still remains in a node, the developer is prompted with the *conflict resolution view* (see Figure 4.7) to decide which correspondence should be selected. In this view, the developer is presented with a node from their source (Figure 4.7a), its context of use in their code (Figure 4.7b), the set of potentially corresponding nodes plus the decision “Do not correspond” (Figure 4.7c), and the source context of the currently selected corresponding node (Figure 4.7d). The decision “Do not correspond” gives the developer the ability to decide that none of the available candidates corresponds with their node.

![Conflict Resolution View](image)

Figure 4.7: The conflict resolution view requests developer input to select a single anti-unifier.

We have found that the default thresholding values result in the selection of correct correspondences in the majority of cases, thereby requiring minimal input from the developer.

For example, the `signature.append(...)` method call on Line 15 in Figure 2.1 has a set of corresponding nodes, including a call to the `signature.append(...)` method on Lines 9 and 19 in Figure 2.2, that pass the inclusion threshold and are within the difference threshold. The developer is prompted for input (Figure 4.7) from which he decides to select “Do not correspond” because the purpose of these method calls is sufficiently different.
4.4 Relevance links

We are interested in creating nodes that insert missing functionality in the paste seed. To do this, we use the parts of the copy seed method that have no correspondences, but that are similar to parts in the copy context class. Therefore, we introduce *relevance links* between nodes in the copy seed method and its context, for nodes referencing identical identifiers.

Relevance links provide a simple mechanism for connecting together a name’s uses and its declaration. From these links Jigsaw can identify dependencies between the copy seed method and its context class. Jigsaw uses relevance links to determine where to insert and how to transform nodes that have no correspondences of their own (see Section 4.5).

The algorithm `CREATE-RELEVANCE-LINKS` determines if two names have the identical binding; if they do, it creates a relevance link between the two names. Each name node contains the set `rlink` of all the name nodes with which it has an identical binding (Lines 4 and 5). For example, the declaration of the name `node` in the original context AST in Figure 4.2 is referenced on Lines 4, 5, 10, and 21 in Figure 2.2. For each of the references, relevance links will be created, as shown in Figure 4.8.

```
Algorithm 10 CREATE-RELEVANCE-LINK(copy), creates a link between name uses and their declarations.

CREATE-RELEVANCE-LINK(copy)
1   for each $k_a \in \text{names}[copy]$
2      do for each $k_b \in \text{names}[copy] \setminus \{k_a\}$
3         do if $\text{binding}[k_a] \equiv \text{binding}[k_b]$
4            then $rlink[k_a] \leftarrow rlink[k_a] \cup \{k_b\}$
5            $rlink[k_b] \leftarrow rlink[k_b] \cup \{k_a\}$
```

\footnote{2} Bindings are computed via Eclipse’s Java Development Tools framework.
4.5 Integration

Once the CAST has been transformed into a representation of the anti-unifier of best fit, we determine the functionality from the copy seed that is absent from the paste seed, by traversing the copy method’s CAST.

If the paste seed was a class, then the entire copy method is interpreted as representing missing functionality. Nodes in the copy method’s CAST that have no correspondence, represent missing functionality to incorporate into the target source code. For copy method nodes that do not have a correspondence but have a relevance link (or several), Jigsaw uses the rele-
vance link to find the copy’s context node that occurs in the anti-unification. This context node establishes the location and form of the “hole” into which the copy method’s node should be inserted. The \textsc{Insert-Missing-Node} algorithm performs the insertion.

\textbf{Algorithm 11} \textsc{Insert-Missing-Node}(a, b), determines the nodes from the reuse source that are to be copied and modified to fit into the developer’s target source.

\begin{algorithm}
\caption{\textsc{Insert-Missing-Node}(a, b)}
\begin{algorithmic}
\State $copy \leftarrow \textsc{Copy-Node}(a)$
\State \textsc{Add-Node-To-Tree}(copy, b)
\If{$\text{rlink}[a] \neq \text{NIL}$}
\State \text{anti-unifier}[a] $\leftarrow$ \text{anti-unifier}[$\text{rlink}[a]$]
\If{\text{anti-unifier}[a] = $\{\}$}
\State \text{loc} $\leftarrow$ \textsc{Find-Location}(\text{rlink}[a], copy)
\State \textsc{Insert-Missing-Node}(\text{rlink}[a], \text{loc})
\Else\quad \textsc{Update}(copy, \text{anti-unifier}[a])
\EndIf
\EndIf\For{each $k \in \text{children}[a]$}
\State \text{do} \textsc{Insert-Missing-Node}(k, copy)
\EndFor
\end{algorithmic}
\end{algorithm}

Jigsaw constructs a new node (Lines 1) to be used in the paste seed method. If the copy method’s CAST node is the root of a subtree, we also create a node for each node of the subtree (Lines 9–10). If the relevance link of the copy method’s node contains an anti-unifier (Line 5), we update the name binding of the new node to that of corresponding node’s (Line 8). The location of the new node is determined by the original node’s positioning relative to correspondence connections in the copy method seed, that is the node is inserted after the last correspondence connection and before the next correspondence connection (Line 6).

For fields and method declarations in the copy context that are called from the copy seed method (i.e., they have a relevance link into the copy method) but that have no anti-unifier (i.e., there is no corresponding field or method declaration in the developer’s target class), we copy these declarations to the end of their respective declaration types in the paste class (Lines 6–8).
From the motivational scenario, we can see two situations in which relevance links are key to correct transformations. (1) The parameter node on Line 3 in Figure 2.2 has relevance links to the statements on Lines 4, 5, and 21 that do not have anti-unifiers; however, the declaration of node corresponds with the declaration of method on Line 13 in Figure 2.1 (described in Section 4.4). Therefore, Jigsaw will copy Lines 4 and 21 from Figure 2.2 and insert them into Figure 2.1, replacing the identifier “node” with the identifier “method” (the result can be seen in Figure 4.9w). (2) In the copy context (Figure 2.2), the two method calls to appendTypeLetter(...) on Lines 14 and 23 have relevance links to the appendTypeLetter(...) method declaration on Line 30; these method calls and the method declaration do not have an anti-unifier. The appendTypeLetter(...) method calls and declaration are thus copied into the paste context (Figures 4.9y and 4.9g).

Once the paste seed’s CAST has been modified to include the missing functionality contained in the copy seed’s method, along with their dependencies, Jigsaw automatically generates the new source code output replacing the developer’s previous source code with the integrated version.

Jigsaw presents its decisions to the developer by augmenting the source code view with four colours (see Figure 4.9). The use of colours allows the developer to more easily identify possible areas of concern. Blue indicates decisions based on relevance-links. Yellow indicates parts of the code that are below the similarity threshold, but due to their context the overall similarity was above the threshold (which could be of potential concern). Green indicates that the statement is identical to that in the copy method. Red is used to indicate type hierarchy conflicts.

From the output produced by the sample scenario (see Figure 4.9) we can identify that: (a) the inheritance hierarchy is the same; (b) all references to node have been replaced with references to method; (c) the if-statement was included based on its usage in computeMethodSignature(...); (d) the signature and parameters variable declarations, along
public class MethodSignVisitor extends ASTVisitor {
    LinkedList<String> slist;
    public LinkedList<String> getListOfMethodSign() {
        // Method implementation...
    }
    public MethodSignVisitor() {
        // Constructor...
    }
    public String getParent(ASTNode node) {
        // Method implementation...
    }
    public boolean visit(MethodDeclaration node) {
        // Method implementation...
    }
}

public String getMethodSign(MethodDeclaration method) {
    // Method implementation...
}

Figure 4.9: Jigsaw’s output based on the motivational scenario of Chapter 2.

with the return statement `signature.toString()` were found to correspond exactly; (e) the for-loop only partially corresponded due to the inclusion of the for-loop body statements from their usage in `computeMethodSignature(...)`; (f) the statements from the copy seed’s for-loop up until the return-statement were included because of their usage in `computeMethodSignature(...)`; and (g) the `appendTypeLetter(...)` method declaration was included because of the method calls in `computeMethodSignature(...)`.
public class Switch {
    public void m2(int i) {
        switch (i) {
            case 1:
                i += 1;
                break;
            case 2:
                i += 2;
                break;
            default:
                i += 3;
                break;
        }
    }
}

public class If {
    public void m1(int i) {
        if (i == 1) {
            i += 1;
        } else if (i == 2) {
            i += 2;
        } else {
            i += 3;
        }
    }
}

Figure 4.10: Semantically equivalent if- and switch-statements.

4.6 Semantics

In our theoretical framework (see Chapter 3) to deal with semantics of our structures, we introduced a theory of equivalence, where we could apply equivalence equations $=_{E}$ to signify if two structures were semantically similar. However, in practice this is more difficult to achieve. To determine if two AST structures are semantically equivalent, we have to abstract the “common” pieces of the two structures and reform them into comparable structures. For example, let’s look back to the semantically equivalent structures $\Delta_{if}(C, B_1, B_2) =_{E} \Delta_{switch}(\Delta_{case_1}(C, B_1), \Delta_{case_2}(\neg C, B_2))$ we discussed in Section 3.4. Similarly to these equivalent structures we can create instances of Java programs that have an if- or a switch-structure (see Figure 4.10). In these two Java programs we have a boolean comparison being done on the parameter int $i$, where if a condition is met $i$ gets incremented by a certain amount.

Starting with the switch-statement, Jigsaw extracts out the switch-condition being passed in, and the values of the case-statements and reforms each of these cases into a boolean expression (i.e., $i == 1$). Jigsaw then extracts the boolean expression contained within the if-statement.
Figure 4.11: Jigsaw’s output based on the example classes from the semantically equivalent statements in the Figure 4.10.

(we consider the default and else to be equivalent in meaning) and compares the boolean expressions to determine if the two statements (i.e., structures) are semantically equivalent. The result output from Jigsaw is that the two statements get marked as having partial correspondence, described in Figure 4.11.

4.7 Multiple MSAs

To create the anti-unifier of best fit we aim to choose the MSA for each node; however, it is possible for each node to have multiple MSAs (discussed in Section 3.3). Jigsaw has two mechanisms to resolve MSAs. One of the mechanisms is the developer’s input, in which the developer decides the best MSA; however, with the possibility of an exponential number of anti-unifiers, it is not ideal to prompt the developer to make a choice for every node in the AST. Therefore it is best to try to address the problem of multiple MSAs when we are calculating the similarity by adding extra assumptions. More precisely, including level information of a node in the AST where slightly more weight is given to those corresponding structures that occur at the same level. We perform this adjustment by increasing the
```java
public class Paste {
    A e = null;

    public void paste(Class b) {
        Object f = new A();
    }
}
```

```java
public class Copy {
    public void copy() {
        A b = new A();
    }
}
```

Figure 4.12: Example of instances of Java programs where the VariableDeclaration in the Copy class anti-unification with the Field, Parameter, and VariableDeclaration from the Paste class can result in multiple MSAs.

If our similarity measure only took into account the type, name, and assignment of our VariableDeclaration, FieldDeclaration, and Parameter we would have to prompt the developer to decide which MSA should be selected. However, we also have the level in which these structures sit in our AST. Therefore we can apply a semantically-based heuristic adjustment and weigh the structures that sit at the same level in our AST as having greater similarity. We perform this adjustment by increasing the size of the corresponding node’s tree in the Paste class (i.e., VariableDeclaration, FieldDeclaration, and Parameter) and the Copy class (i.e., VariableDeclaration) by 1 in the COMPARE-NON-LEAVES algorithm (in Section 4.2) and incrementing the best value of the comparison of the two VariableDeclarations by 1 if the structures sit at the same-level. Applying Jigsaw to our example from above, we can see
the adjustment results in an elimination of multiple MSAs, where the `VariableDeclaration` in the `paste` method is selected as the anti-unifier of best fit shown in Figure 4.13.

Multiple MSAs can also occur with the nesting of similar structures. So we apply this technique to `if`-structures and loop-structures that can contain similar nested structures with them, to help distinguish between nested structures to select the anti-unifier of best fit.

![Figure 4.13: Jigsaw’s output based on the example classes from the multiple MSAs scenario in Figure 4.12.](image)

4.8 Summary

We have implemented an approach for semi-automating small-scale source code reuse within a tool called Jigsaw. Jigsaw identifies correspondences between two ASTs through the application approximated higher-order anti-unification modulo theories. We have shown how concrete implementation of the set of theories $\mathcal{E}$ through our similarity measure can be applied to anti-unify the two structures. Jigsaw evaluates the quality of the anti-unifiers by a greedy selection algorithm to form an anti-unifier of best fit; this is used to create a version of the reused source code that integrates into the developer’s target system.
Chapter 5

Evaluation

To evaluate our approach and tool, we conducted two empirical studies. The first (Section 5.1) was an experiment to address the question: “Can Jigsaw mimic the results of suspected, real small-scale reuse tasks discovered in an industrial system, to produce an integration of comparable quality?” The second (Section 5.2) was a pair of case studies to address the questions: “Can an industrial developer use Jigsaw to complete small-scale reuse tasks?”, “Do industrial developers find Jigsaw a potentially useful tool?”, and “What usability changes would improve the efficacy of Jigsaw in an industrial context?”

5.1 Experiment

To identify cases of small-scale reuse within methods we used a clone detection tool [VD03]. The CCFinder\(^1\) clone detection tool [KKI02], was applied to the Eclipse Java Development Tools (JDT) and the Eclipse Annotation Processing Tool (APT).

CCFinder identified 49,436 clone pairs (in 4,473 class files). We filtered the results to show only classes where no more than 3 clones were present (resulting in 1,230 class files), because classes that have a large number of code clones are likely candidates for more heavyweight reuse techniques such as abstraction-based refactorings or complete redesigns. For the remaining code clones, we manually selected the first 15 cases that we encountered (by systematically traversing the results) that involved clones between methods and where the clone size was at least 4 lines of code (LOC). The resulting 15 test cases are described in Table 5.1.

\(^1\)Version 20071121 with default settings.
<table>
<thead>
<tr>
<th>Test case</th>
<th>Classes</th>
<th>Method name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>apt.core.internal.declaration.ConstructorDeclarationImpl</td>
<td>toString()</td>
</tr>
<tr>
<td></td>
<td>apt.core.internal.declaration.MethodDeclarationImpl</td>
<td>toString()</td>
</tr>
<tr>
<td>2</td>
<td>apt.core.internal.declaration.ASTBasedAnnotationElementDeclarationImpl</td>
<td>getReturnType()</td>
</tr>
<tr>
<td></td>
<td>apt.core.internal.declaration.ASTBasedMethodDeclarationImpl</td>
<td>getReturnType()</td>
</tr>
<tr>
<td>3</td>
<td>apt.ui.internal.preferences.AptConfigurationBlock</td>
<td>editOrAddProcessorOption(...)</td>
</tr>
<tr>
<td></td>
<td>apt.ui.internal.preferences.TodoTaskConfigurationBlock</td>
<td>doTodoButtonPressed(...)</td>
</tr>
<tr>
<td>4</td>
<td>internal.core.search.matching.PackageReferenceLocator</td>
<td>resolveLevel(...)</td>
</tr>
<tr>
<td></td>
<td>internal.core.search.matching.TypeReferenceLocator</td>
<td>resolveLevel(...)</td>
</tr>
<tr>
<td>5</td>
<td>internal.compiler.ast.NormalAnnotation</td>
<td>printExpression(...)</td>
</tr>
<tr>
<td></td>
<td>codeassist.complete.CompletionOnAnnotationMemberValuePair</td>
<td>printExpression(...)</td>
</tr>
<tr>
<td>6</td>
<td>codeassist.complete.CompletionOnMarkerAnnotationName</td>
<td>resolveType(...)</td>
</tr>
<tr>
<td></td>
<td>codeassist.complete.CompletionOnQualifiedTypeReference</td>
<td>getTypeBinding(...)</td>
</tr>
<tr>
<td>7</td>
<td>internal.compiler.codegen.CharArrayCache</td>
<td>rehash()</td>
</tr>
<tr>
<td></td>
<td>internal.compiler.codegen.ObjectCache</td>
<td>rehash()</td>
</tr>
<tr>
<td>8</td>
<td>internal.compiler.util.COMPOUNDNAMEVECTOR</td>
<td>remove(...)</td>
</tr>
<tr>
<td></td>
<td>internal.core.TypeVector</td>
<td>remove(...)</td>
</tr>
<tr>
<td>9</td>
<td>internal.compiler.batch.ClasspathDirectory</td>
<td>doesFileExist(...)</td>
</tr>
<tr>
<td></td>
<td>internal.core.builder.ClasspathDirectory</td>
<td>doesFileExist(...)</td>
</tr>
<tr>
<td>10</td>
<td>internal.core.search.matching.InternalSearchPattern</td>
<td>findIndexMatches(...)</td>
</tr>
<tr>
<td></td>
<td>internal.core.search.matching.AndPattern</td>
<td>findIndexMatches(...)</td>
</tr>
<tr>
<td>11</td>
<td>internal.core.CreateCompilationUnitOperation</td>
<td>getSchedulingRule()</td>
</tr>
<tr>
<td></td>
<td>internal.core.CommitWorkingCopyOperation</td>
<td>getSchedulingRule()</td>
</tr>
<tr>
<td>12</td>
<td>internal.jdwp.JdwpReplyPacket</td>
<td>errorMap()</td>
</tr>
<tr>
<td></td>
<td>internal.spy.JdwpReplyPacket</td>
<td>errorMap()</td>
</tr>
<tr>
<td>13</td>
<td>internal.junit.ui.CopyFailureListAction</td>
<td>run()</td>
</tr>
<tr>
<td></td>
<td>internal.ui.callhierarchy.LocationCopyAction</td>
<td>run()</td>
</tr>
<tr>
<td>14</td>
<td>internal.corext.refactoring.rename.RenameResourceProcessor</td>
<td>initialize(...)</td>
</tr>
<tr>
<td></td>
<td>internal.corext.refactoring.rename.RenameSourceFolderProcessor</td>
<td>initialize(...)</td>
</tr>
<tr>
<td>15</td>
<td>internal.ui.refactoring.reorg.TypeSourceTransfer</td>
<td>javaToNative(...)</td>
</tr>
<tr>
<td></td>
<td>internal.corext.refactoring.reorg.JavaElementTransfer</td>
<td>javaToNative(...)</td>
</tr>
</tbody>
</table>

Table 5.1: Method pairs used as test cases; all are contained in the `org.eclipse.jdt` package.

With the 15 Java class pairs selected, we investigated the methods identified by the clone detector to infer the likely direction of the original reuse. We used the inclusion of source code that was not part of the code clone within one of the methods as an indication of the intended paste seed. For methods where this information is not present, we chose the direction randomly (test cases 2, 7 and 8). The reuse direction that we used is represented in Table 5.1 for each test case, where the first method is the original context (copy seed) and the second method is the developer’s target context (paste seed). The paste seed was then modified; duplicate code between the seeds in which there did not exist a dependency from the included source code was removed. For example, the duplicate code in the paste seed method `toString()` from test
case 1 are lines 2–3 and 17–25 in Figure 5.1. Analyzing the duplicated LOC, we can identify a dependency on lines 2 and 3 from the variable declaration on line 4; therefore, only lines 17–25 were removed.

The copy & paste seeds in test case 7 contain a method call to a duplicate method declaration. The existence or non-existence of the method could affect Jigsaw’s integration; therefore, we split the test case into two: test case 7 contained the method declaration in the paste seed class; in test case 7*, the method declaration was removed from the paste seed class.

Jigsaw was then supplied the copy & paste seeds. If Jigsaw’s conflict resolver prompted for information, the original method was used to determine the intended action.

5.1.1 Results

The analysis of the output has been broken down into three categories: conflict resolution, correspondence, integration, and time. The results of this analysis are shown in Table 5.2.

The number of times that Jigsaw’s conflict resolver (see Section 4.3) prompted for developer input, is indicated by the conflict resolution column. Test case 1, 5, 7, 7* and 10 were situations similar to the situation described in Section 4.3 where there was a set of method calls that differed by a parameter. Test case 10 also contained multiple identical \texttt{if}-statements that were being used for different purposes at different locations in the copy & paste seeds.

We divided correspondence into two categories: the number of correct correspondence connections Jigsaw formed between the copy & paste seeds, and the number of correspondence connections Jigsaw formed between the copy & paste seeds that did not fit the context of use. We determined the context of use by comparing Jigsaw’s result to the actual method. We also report the percentage of the total number of correct correspondence connections between the copy & paste seeds. In test cases 1, 4, and 10, Jigsaw marked an identical statement in the copy seed as corresponding with a statement in the paste seed that was being used for a

\footnote{The workstation evaluating the Jigsaw plug-in was a Macbook 2GHz Core 2 Duo with 2GB RAM.}
public String toString() {
    final String buffer = new StringBuilder();
    final IMethodBinding methodBinding = getDeclarationBinding();
    final ITypeBinding[] typeParams = methodBinding.getTypeParameters();
    if (typeParams != null && typeParams.length > 0) {
        buffer.append('<');
        for (int i = 0; i < typeParams.length; i++) {
            if (i != 0) buffer.append(', ');
            buffer.append(typeParams[i]);
        }
        buffer.append('>');
    }
    if (methodBinding.getReturnType() != null)
        buffer.append(methodBinding.getReturnType().getName());
    buffer.append(methodBinding.getName());
    buffer.append('(');
    int i = 0;
    for (ParameterDeclaration param : getParameters()) {
        if (i++ != 0) buffer.append(', ');
        buffer.append(param);
    }
    buffer.append(')');
    return buffer.toString();
}

Figure 5.1: Paste seed from test case 1 (org.eclipse.jdt.apt.core.internal.declaration.MethodDeclarationImpl.toString()).
different purpose. Test case 10 also marked a highly similar `if`-statement in the copy seed as corresponding with an `if`-statement that was being used for a different purpose in the paste seed. The effect of these incorrect correspondences caused the `return`- and `if`-statements (the body of the `if`-statement is treated separately, as described in Section 4.2) to be not integrated into the paste seed. In test case 15, two statements were placed before a statement where in the original method the statements were to come after. In that case there did not exist sufficient contextual information for our heuristics to guide the integration of those two statements into the correct placement.

We compared Jigsaw’s resulting integration with the original paste seed method as the baseline to calculate the percentages for the correctly integrated LOC. If a line of code was not integrated (i.e., in test cases 1, 4, and 10) or the line of code was not integrated properly (i.e., test case 15), then it was counted as incorrect.

Jigsaw’s execution time was recorded in seconds for all test cases. The test cases in which the developer’s input was required, we excluded the developer’s input time from the execution time. The tool’s execution time for the test cases ranged from 0.1 to 5.2 seconds.

5.1.2 Lessons learned

Jigsaw was successful in producing integration results of equal quality compared to the original paste seeds in 12 out of the 16 test cases.

In test cases 7 and 7*, the existence/non-existence of the method dependency had no distinguishable effect on the integration. In test case 7* where the correspondence did not exist, the relevance link was used by Jigsaw to correctly infer its addition to the paste seed class.

The application of higher-order anti-unification for determining correspondence eliminated the mismatch and conceptual disconnect errors that we reported in our previous application of simple anti-unification [CCWD07]. The time cost accrued to perform the high-order anti-unification and the integration of the reuse source in the developer’s target was inconsequential.
Table 5.2: Results from applying Jigsaw to the 15 (+ 1) test cases. Times marked thus (†) do not include the time required for interaction with the developer in cases where developer input was needed.

In the 4 cases (i.e., test cases 1, 4, 10, and 15) where Jigsaw’s integration results were not of a quality equal to that of the original paste seed, contextual errors where found and announced to the developer. Jigsaw correctly identifies similar elements and forms a correspondence connection, however, during the integration step; Jigsaw does not take into account the node’s context with respect to its parent node container (e.g. an assignment statement contained in a if-statement). Conceptual information is not captured by the language, therefore, in many situations the information needed to determine a node’s context does not exist. The results from the evaluation show that when the context errors occur, their effect on the integration is minimal: in the evaluation the effect of the error was 2 LOC (test cases 10 and 15) integrated incorrectly in the worst case.
5.2 Case studies

The participants recruited for the case studies were 2 developers (S1 and S2), with 3 and 8 years of industrial experience respectively. The developers are experienced in the Java programming language and have experience working with the Eclipse IDE. The developers were assigned 3 unique, small-scale reuse tasks each, selected randomly from the cases identified in Section 5.1. The developers were given the same copy & paste seeds that were determined in Section 5.1. Participant S1 received Tasks 1, 3, and 7*, and Participant S2 received Tasks 2, 4, and 5 from Table 5.1. The developers were given a 5 minute training session on how to operate Jigsaw using the motivational scenario from Section 2. For each task, the developer was given a brief description of the reuse scenario. The two developers applied Jigsaw to their assigned reuse tasks and were interviewed about their interactions with the tool.

5.2.1 Results

The developers found that Jigsaw supported them in performing small-scale reuse tasks. The tool removed them from the conceptually straightforward aspects of reuse and allowed them to focus on the higher-level details: “Forces me to think: why I am reusing this method?” [S1] and “Do I really need this block of code?” [S1]. The use of colour highlighting allowed the developers to identify and confirm their thoughts about the reuse: “[It] makes me aware of the use of inheritance and dependencies the method has” [S2]. The developers for tasks 1 and 4 identified the missing return-statement through the colour highlighting (yellow) of an unexpected return-statement in the paste seed. Participant S1 also reported that during task 7* the tool made him aware of a method call by including it with the integration: “[Jigsaw] helped me identify a method call that I did not realize was missing”.

The developers thought the conflict resolver was sufficient for single statements; however, for larger blocks of code they would have liked the ability to preview the outcome of their
decisions. The developers found the ability to quickly “undo” the integration helpful when working with the conflict resolution view; the developers were able to try different decisions: “the undo option makes it easier for me to investigate different options” [S1].

The industrial developers confirmed that these were the types of cases in which they would perform small-scale reuse. Participant S1 reported that his original approach to small-scale reuse was “mashing it to fit”, whereas Jigsaw allowed him to “focus on [his] intent”. Participant S2 would carefully analyze any block of code before reusing, but found Jigsaw helpful in removing the menial tasks and confirming his thoughts: “I would use [Jigsaw] to confirm this is the action I wanted.”

The developers were able to use Jigsaw to produce results equivalent to the original paste seed, with no knowledge of the expected outcome. Furthermore, the results from the case studies show that Jigsaw can assist an industrial developer perform small-scale reuse tasks. Jigsaw removed the menial aspects of the reuse, while including/highlighting source code overlooked by the developer. The developers found the tool improved their approach to small-scale reuse by focusing and/or confirming their intentions. The developers also provided valuable information for improving the way conflict resolution is handled.

5.3 Summary

Our evaluation is based on three questions: “Can Jigsaw mimic the results of suspected, real small-scale reuse tasks discovered in an industrial system, to produce an integration of comparable quality?”, “Can an industrial developer use Jigsaw to complete small-scale reuse tasks?”, “Do industrial developers find Jigsaw a potentially useful tool?”, and “What usability changes would improve the efficacy of Jigsaw in an industrial context?”

To address these questions we performed two empirical studies to evaluate Jigsaw’s potential to integrate reused source code into the developer’s target system. For the first experiment,
to identify cases of small-scale reuse we used a clone detector tool, in which we found 15 class pairs from industrial code. From the 15 class pairs, we were able to derive 16 test cases. Jigsaw’s integration resulted in equal quality compared to the original paste seeds in 12 out of the 16 test cases. The second experiment involved two industrial developers using the Jigsaw tool to perform small-scale reuse on 3 test cases found in the first experiment. The developers in the case studies were able to use Jigsaw to produce results equivalent to the original paste seeds, with no knowledge of the expected outcome and found that Jigsaw focused their attention on the most difficult issues of a small-scale reuse task.
Chapter 6

Related Work

In this chapter we discuss related work to our problem of small-scale reuse (Section 6.1), copy & paste as a means of reuse (Section 6.2), and determining correspondences (Section 6.3).

6.1 Reuse

A variety of reuse research has focused on the construction of software components and libraries for reuse. While several approaches have advocated refactoring code into reusable application programming interfaces (APIs), there are many situations in which these techniques are too heavyweight or simply not practical for small-scale source code reuse. Krueger [Kru92] notes that the construction of software components and libraries from scavenged small-scale source code requires significant manual effort for the developer, outweighing the benefits gained from reuse. It has also been shown that reused source code generally must be modified in some way to work within its new context, even in organizations with explicit reuse programs in place [Sel05]. Reuse tasks can be divided into two phases: location and integration.

6.1.1 Locating examples of reuse

Several approaches focus on identifying source code use examples. Basili et al. [BCC92] describe an approach for extracting reusable components from an existing source code based on a set of metrics, meaning that the developer may not get to reuse the particular functionality he desires. Michail describes the CodeWeb tool [Mic01] that applies data mining association rules to determine reuse patterns. Xie and Pie [XP06] describe a data mining approach using an API’s usage history to identify call patterns. Holmes et al. have created the Strathcona tool, to search for code that is similar to some input code skeleton for the sake of seeking examples.
of how to use an API [HWM06]; Strathcona extracts the structural context (a set of facts such as the calls being made and the types being referenced) from code highlighted by the developer and compares it against the structural context of code in a repository. None of these approaches support the developer in the integration phase of reuse.

6.1.2 Integration

Relatively little work has considered the integration phase. Notable exceptions are approaches requiring formal specifications to perform the integration phase (e.g., [GH91, YS97]); in our context, such formal specifications cannot be expected to be pre-existing and the cost of developing them would likely outweigh the benefit from the tool support. Gilligan [HW07] allows developers to plan medium- to large-scale reuse tasks at the integration phase, and Procrustes [HW08] supports the semi-automated enactment of these plans; planning and enactment of these tasks still requires significant manual effort from the developer and this approach is likely not cost-effective for small-scale tasks.

Generalization-based refactorings [TKB03], abstract out the common code for use in a general context in which specification can then be later applied. While these refactorings are often cited as a means to perform small-scale reuse they are not applicable in situations where: the developer does not have ownership of the code; the refactoring exposes functionality that violates the system’s design; logical constraints exist that make the creation of an abstraction impossible [TBG04, KSNM05].

Solar–Lezama et al. [SLTB+06, SLAT+07] developed Sketch, a language for the creation of partial implementations (sketches) and separate specification of the desired, missing functionality. Sketch uses a combinatorial search algorithm at runtime to find the appropriate sketch that meets the requirements of the specification to fill in the desired, missing functionality. It is unclear that the required investment required from a developer to create a sketch to reuse a small code fragment would be cost-effective relative to the benefit.
6.2 Copy & paste

Developers perform “copy & paste” as a means to reuse source code in situations where abstraction based methods are not applicable. Kim et al. [KBLN04] report that developers use “copy & paste” as a means of reuse on average four times per hour. Modifications to the original (i.e., copy) source often requires parallel updates to the code clone aspect of the reuse in the paste source, and vice versa [KSNM05]. Several researchers have developed approaches to assist developers in evolving these clones. Duala-Ekoko and Robillard [DER07] describe an approach for tracking evolving code clones that are mapped across a system, with prototype support for their simultaneous editing. CReN [JH07] tracks copy & paste clones and applies a set of rules to identify inconsistencies in renaming; it automatically renames the identifiers in the clone group. In contrast, Jigsaw integrates missing functionality from an originating system (a method and its dependencies) into the developer’s target system.

While those approaches assist developers in evolving the code clones, the developer is still left with the manual task of integrating the reuse source into their target source. An approach by Edwards [Edw05] investigates copying as a means of programming through a tool called Subtext. Programs are created by the copying of nodes in a tree structure that create new child nodes (or subtrees) which are then are modified by a developer to provide a new functionality in their programs. Similarly, the developer indicates to Jigsaw the reuse source code that they want to reuse; Jigsaw then performs a generalization of two ASTs supplied to determine the nodes that are to be copied and modified into the developer’s target context. Jigsaw offers several advantages to Subtext: Jigsaw does not require developers to learn a new paradigm; Jigsaw modifies the reuse source to fit into the developer’s context; and Jigsaw offers the developer the ability to reuse a vast amount of existing functionality.
6.3 Correspondence

Jackson and Ladd [JL94] describe an approach that uses semantic information to identify differences between two sources. Their approach provides a guide to determine the correspondence of two elements of similar semantic context; however, the developer still manually performs the integration. Apiwattanapong et al. describe the JDiff tool [AOH04, AOH07] that uses a top–down approach to create pairs of matched nodes in an AST. Their approach allows them to systematically identify cases of addition, deletion, and modification at the class- and interface-level, the method-level, and the node-level. Our approach uses a bottom–up approach to determine the correspondence between nodes. In our problem context, the details contained at the bottom levels of the AST are needed for the modification aspects of the integration.

Our previous work on the Breakaway tool [CCWD07] focused on providing a detailed correspondence view between two source inputs. Breakaway automatically forms a generalized view through a two-pass lexically-greedy correspondence algorithm that does not consider correspondence between one and multiple elements and ignores language-specific semantic information, an important consideration in small-scale reuse tasks. The problem of small-scale reuse is more complex, as the integration phase is also influenced by the non-corresponding structural elements that exist between two source entities. Our earlier research has required significant extension to address this problem.

Several areas of software engineering (e.g., profile propagation, regression testing, software version merging, multi-version program analysis, origin analysis, etc) look to determining similar elements between two versions of a program as a fundamental building block [KN06]. These areas employ a vast set of tools [Tic84, WPM00, GZ05, KPW05] with varying levels of granularity in their similarity measures to determine the correspondence between elements, that often requires the application of multiple tools to get a good level of matching [KN06]. Similar to the set of tools, Jigsaw uses a similarity measure to find correspondences between
elements, however, differs by determining the correspondences at the structural level. Origin analysis [GZ05] and multi-version program analyses [KN06] look to identify the changes between versions of software, similarly, Jigsaw provides a mechanism for abstracting the “common” pieces between two sources. However, unlike the problems of origin analysis and multi-version program analyses were the assumption exists that versions of software are related, our problem of small-scale reuse requires Jigsaw to be more robust as we cannot make the same assumptions about two sources supplied as input.

6.4 Summary

Little work has focused on the integration phase of reuse; this has looked at the integration phase, often require developers to have foresight into how the existing source code will be reused, and the reuse must be large enough to benefit from the cost developer must invest. However, analysis of developer work habits have shown that developers perform small-scale reuse task on a regular basis in their development activities where the approaches mentioned above are likely too heavyweight to be applicable. Some work has looked into helping developer’s maintain clones between copy & paste sources, but still do not assist developers in performing the integration phase of the reuse.
Chapter 7

Discussion

In this chapter, we discuss the validity of our evaluation and case studies (Section 7.1), a number of remaining issues with Jigsaw (Section 7.2), and the application of small-scale reuse by means of copying (Section 7.3).

7.1 Threats to validity

Our goal was to discover the strengths and weaknesses of the approach and our current tool support. Therefore we undertook an experiment and case studies to provide initial results of the promise of our approach. The tasks chosen with the use of a clone detector were from two parts of a single, large-scale system comprising 4,473 classes. To limit the bias towards highly similar classes where a more heavyweight reuse technique would likely be applied, the clone detector results were filtered to 1,230 class files. By filtering the results we selected the first 15 examples, spanning a range of complexity. Note that the idea of leveraging a clone detector to reconstruct the evolution of a system has precedents in the literature [VD03].

The experiment involved one of the authors performing the integration. The results from each integration were compared with the original, real methods as the baseline. When the author was prompted by the conflict resolver, his input could have been biased in favour of the tool. To avoid this, he reviewed the original method to mimic the developer’s intent. To avoid biasing the tool’s execution time by the author’s knowledge of the conflict resolutions, we only recorded Jigsaw’s execution time and excluded the time taken by the author to resolve the conflicts. While we cannot claim that conflict resolution would never be onerous, we do claim that the developer could make the pragmatic decision to manually deal with some
problems. Our industrial case studies indicate that conflict resolution is not especially onerous in the cases examined, though usability improvements to our tooling were suggested (and are currently being pursued by us).

The case study participants were two developers with a variety of industrial experience from different organizations. The developers’ tasks were chosen randomly from the set of test cases determined for the experiment. The number of participants was small; however, our intention was to explore their use of the tool in detail. The developers were able to produce results equivalent to the original paste seed, with no knowledge of the expected outcome.

7.2 Jigsaw’s output

We can only guarantee that Jigsaw will produce syntactically correct source code, not that the result is compilable or correctly executable because of issues of type correctness, inheritance structures, and node ordering that our approximation approach does not handle perfectly.

7.2.1 Type correctness and inheritance hierarchy

Jigsaw does not guarantee type correctness as it uses only basic typing information to determine if two type nodes correspond; this was a design decision to tradeoff soundness of analysis for speed and flexibility. As a result, situations can occur where a decision is made by either Jigsaw or the developer to mark two different types as corresponding that, in fact, are not.

Non-executable code can occur when nodes that have a relevance link access information contained in the inheritance hierarchy in the copy method, but the relevance link has no correspondence with the paste method. If the paste class does not have an inheritance hierarchy established, we update the inheritance hierarchy to be identical to that of the copy class, potentially introducing an object that the developer does not have access to. If the paste class has an inheritance hierarchy that is not equivalent to that of the copy class, comments are inserted regarding the potential conflict.
7.2.2 Node ordering

The placement of nodes (see Section 4.5) that do not correspond is guided by those nodes that do. This creates a possibility that the correct ordering of nodes inside a method will not be maintained. For example, if we had two sequences of nodes \{C, A\} and \{A, B, C\}, with elements C and A corresponding, Jigsaw’s approach to ordering would result in the new set \{B, C, A\}. In situations where B is dependent on being preceded by A, the resulting node ordering could result in non-executable code.

7.3 Small-scale reuse

The modification and integration of reused source code into the developer’s target requires many trivial steps. Brooks describes these trivial steps as accidental to the software development process, “those difficulties that . . . attend its production but are not inherent” [FPB87]. These trivial steps are accidental rather than essential to the difficulty of the problem of reusing source code. By relieving the developer of these accidental tasks, Jigsaw enables them to pay closer attention to the difficulties inherent in the nature of reuse.

Small-scale reuse that employs copying and integrating fragments of source code is often referred to as a bad form of reuse primarily because it introduces code clones and promotes bad programming practices.

The first conjecture that small-scale reuse that employs copying leads to the introduction of code clones which violate the principles of abstraction, is based on a fallacy. There are often present logical or conceptual constraints in which reuse by abstraction is impossible [TBG04, KSNM05]. Developers perform copy and paste reuse on average four times per hour [KBLN04] which allows us to postulate that potential effects code clones could have on their system (e.g., updating clones consistently) have an acceptable cost with respect to benefits reuse offers.
The second conjecture that small-scale reuse that employs copying promotes bad programming practices can be invalidated by drawing a parallel to the practice of object oriented programming (OOP). Object orientation harnesses our intuitive abilities to formally define the world in terms of objects; however, research into the application of OOP is reporting that developers are finding it difficult to learn and apply [Arm06, Hol99]. Our intuitive ability often conflicts with OOP causing developers to misuse the paradigm to their own detriment [HL08]. Because of the misuse of the OOP, the community does not promote it as a bad practice, but on the contrary, encourages the creation of tools to better support developers using OOP. The goal of Jigsaw is the same; to promote the proper application of small-scale reuse by providing the developer with a mechanism for focusing the developer on the conceptually complex issues, highlighting potential areas of concern, and removing them from the conceptual simple modifications that can distract from the potential problems that the reuse may present.

7.4 Anti-unification beyond software development

Anti-unification is a robust theoretical framework for abstracting two structures. The types of structures that we can perform anti-unification on are not limited to the structures found in the software engineering domain. Therefore, anti-unification can be used to create abstractions for any type of structure. The difficulty with anti-unification is creating an abstraction (i.e. anti-unifier) with meaning that is not just variable substitution.

We have shown that the more knowledge that can be harnessed from the structures the more meaningful the abstraction. A key to this has been the addition of theories to deal with concepts not represented in the structures directly but is known about them (e.g., language semantics, varying structure sizes, etc.). Though the theories outlined in this thesis were very successful, they may not be appropriate for structures found in other domains. Therefore one has to develop a set of theories (and combinations thereof) that are appropriate for their needs.
The particular form of anti-unification we utilize is higher-order anti-unification modulo theories which has been proven to be undecidable. However, this work demonstrates how the application of an approximated form of higher-order anti-unification modulo theories can be used successfully to determine structural correspondence. This work also shows how the structural correspondence can be used to inform on related structures that have no structural correspondence.

7.5 Summary

Our goal of our evaluation presented in Chapter 5 was to discover the strengths and weaknesses of the approach and our current tool support. When determining the test cases for our evaluations, to limit the bias towards highly similar classes, we filter the results from the clone detector. When performing the integration for test cases that were prompted by the conflict resolver, the author used the original method to mimic the developer’s intent. The resulting integration from each of the test cases were compared with the original, real methods as the baseline.

Jigsaw will always produce syntactically correct source code, however, because our approach is approximation we cannot guarantee that the resulting output from Jigsaw is compilable or correctly executable.

The goal of Jigsaw is to promote the proper application of small-scale reuse by providing the developer with a mechanism for focusing the developer on the conceptually complex issues, highlighting potential areas of concern, and removing them from the conceptual simple modifications that can distract from the potential problems that the reuse may present.
Chapter 8

Conclusion

Developers perform small-scale reuse regularly during their development activities. However, the integration of source code during reuse places a heavy burden on developers, drawing their attention away from the conceptually complex questions and towards the menial tasks of the integration.

We have presented an approach for semi-automated small-scale source code reuse by means of structural correspondence. We have developed Jigsaw, a prototype tool that identifies correspondences through anti-unification, evaluates their quality, and uses the best candidate to create a version of the reused source code that integrates into the developer’s target system. The problem of anti-unifying code in this manner is known to be undecidable in general; the goal-orientation of our tool allowed us to approximate the needed anti-unification sufficiently to support the developer in his task. Our approach uses syntactical and semantical information about the source code being manipulated and its surrounding context; this information is leveraged heuristically during the anti-unification process.

We have demonstrated Jigsaw’s potential to integrate reused source code into the developer’s target system through an experiment and case studies involving 2 industrial developers. The experiment found that Jigsaw was successful in producing integration results of equal quality compared to the original paste seeds in 12 out of the 16 test cases. The developers in the case studies were able to use Jigsaw to produce results equivalent to the original paste seed, with no knowledge of the expected outcome. The case studies also highlight Jigsaw’s potential to focus the developer’s attention to the most difficult issues in a small-scale reuse task.
The contributions of this thesis are:

- An approach for implementing approximated higher-order anti-unification modulo theories for determining structural correspondences between two AST structures.
- A similarity measure that determines equivalency between different structures that have similar semantic meaning.
- An approach for semi-automatically performing small-scale source code reuse via structural correspondence.

We discuss remaining issues and how our future work will be directed to address these issues.

*Contextual semantics.* Relevance links were used to address issues with child nodes containing dependencies that may sit outside a parent node’s visibility. From the experiments we identified cases (e.g., test cases 1, 4, and 10) where the correspondence of a node contained within an *if*-statement was incorrect because Jigsaw was unaware of the node’s contextual use. We will further investigate ways to leverage the semantics of the language to identify cases where the context of a correspondence pair does not match. Data flow analysis could be harnessed to identify the occurrences of incorrect node ordering, but the complexity of analyses must necessarily be limited to maintain a practical rate of interaction.

*Conflict resolution.* The case studies found that the conflict resolution view did not scale well from a single statement to a block of code. The developers also wanted greater control over the correspondence connections. From that feedback, we are currently investigating the creation of a view that will give the developer more control over the integration through a preview mechanism. The integration previewer would allow a developer to modify the correspondence connections to get instant feedback on the effect that their choice would have on the integration.
Type correctness and inheritance hierarchy. To improve Jigsaw’s knowledge of typing and inheritance hierarchy structures, we will investigate the use of more enhanced type checking approaches such as the type constraint rules described by Tip et al. [TKB03]. The proposed integration previewer would also allow us to harness developer knowledge of the typing and inheritance structures.

Extensions. Any application that is involved in the location phase (e.g., [BCC92, Mic01, HWM06, XP06]) of software reuse could be improved by Jigsaw’s underlying framework. An example is the Strathcona example recommender [HWM06]. Strathcona returns a list of API usage examples, derived from source code stored in a repository. Jigsaw could be used to create previews of the reuses within the developer’s source code.

There are several areas of software engineering (e.g., profile propagation, regression testing, software version merging, multi-version program analysis, origin analysis, etc.) that employ tools to determine the correspondence between two elements. Jigsaw’s ability to determine correspondence between two elements by means of their structures could be of benefit to these areas. To determine what possible benefit Jigsaw could be, we will look into performing a detailed comparison against existing tools (e.g., bdiff [Tic84], BMAT [WPM00], Zou and Godfrey’s hybrid approach [GZ05], JDiff [AOH04, AOH07], etc).
Bibliography


Appendix A

Ethics Approval

CERTIFICATION OF INSTITUTIONAL ETHICS REVIEW

This is to certify that the Conjoint Faculties Research Ethics Board at the University of Calgary has examined the following research proposal and found the proposed research involving human subjects to be in accordance with University of Calgary Guidelines and the Tri-Council Policy Statement on "Ethical Conduct in Research Using Human Subjects". This form and accompanying letter constitute the Certification of Institutional Ethics Review.

File no: 5146
Applicant(s): Rylan R. Cottrell
Department: Computer Science
Project Title: Using Structural Correspondence to Infer the Developer’s Context for Small Scale Code Reuse
Sponsor (if applicable):

Restrictions:

This Certification is subject to the following conditions:

1. Approval is granted only for the project and purposes described in the application.
2. Any modifications to the authorized protocol must be submitted to the Chair, Conjoint Faculties Research Ethics Board for approval.
3. A progress report must be submitted 12 months from the date of this Certification, and should provide the expected completion date for the project.
4. Written notification must be sent to the Board when the project is complete or terminated.

Janice Dickin, Ph.D, LLB,
Chair
Conjoint Faculties Research Ethics Board

Distribution: (1) Applicant, (2) Supervisor (if applicable), (3) Chair, Department/Faculty Research Ethics Committee, (4) Sponsor, (5) Conjoint Faculties Research Ethics Board (6) Research Services.

2500 University Drive N.W., Calgary, Alberta, Canada T2N 1N4 • www.ucalgary.ca