Semi-automating Small-Scale Source Code Reuse via Structural Correspondence

Rylan Cottrell, Robert J. Walker, Jörg Denzinger
Laboratory for Software Modification Research
Department of Computer Science
University of Calgary
{cottrell, rwalker, denzinge}@cpsc.ucalgary.ca

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 paper presents an approach that uses the developer's context to help integrate the reused source code into the developer's own 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.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.6 [**Software Engineering**]: Programming Environments—*Interactive environments*; D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Design, Algorithms, Human Factors.

Keywords

Small-scale source code reuse, structural correspondences, semi-automation, Jigsaw.

SIGSOFT 2008/FSE-16, November 9–15, Atlanta, Georgia, USA

1. INTRODUCTION

Reuse is a goal that has traditionally been approached through explicit abstraction and pre-planning [7]. 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 [3]. However, in practice, developers also need to reuse smaller pieces of functionality, on the scale of individual methods and classes [9]. 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 [16, 15].

Small-scale reuse tasks are performed by developers regularly during their development activities [9]. For example, understanding how to interact with an application programming interface (API) often begins with finding examples of its use [20, 11]. 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.

Reuse tasks can be divided into two phases: location and integration of source code. Much previous research has focused on the location phase (e.g., [2, 18, 11, 24]); relatively little work has considered the integration phase. Notable exceptions are approaches requiring formal specifications (e.g., [8, 25]); 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., [9, 10]), which is likely too heavyweight for small-scale tasks. Our previous research [5] has considered how two classes can be compared to determine their detailed structural correspondences for generalization tasks; 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.

Our approach proceeds in three steps. First, it determines candidate correspondences between terms (e.g., expressions, statements, declarations) in the original system and the target system, using a measure of similarity that considers both structural information about the terms and how they are used within their context. Terms that have a similarity value above a threshold are considered corre-

spondence candidates. Second, when a particular term is a candidate to correspond with more than one other term, 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 terms 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.

We have implemented 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.

To evaluate our approach, we have conducted two empirical studies. In the first, we 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.

The remainder of the paper is structured as follows. Section 2 motivates the problem through a small-scale source code reuse scenario. Section 3 details related work and how it does not adequately address the problem. Section 4 describes our approach for performing small-scale reuse through the use of structural correspondence. Section 5 presents our evaluation of the Jigsaw tool and our approach. Section 6 discusses remaining issues, theoretical foundations, and future work.

The contribution of this paper is an approach for semi-automatically performing small-scale source code reuse.

2. MOTIVATIONAL SCENARIO

Consider a developer trying to generate a Java method signature from an abstract syntax tree (AST) using the Eclipse Java Development Tools[1] (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 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 1a, lines 18–20).

The API documentation does not provide any immediate guidance, so he turns to Strathcona [11], 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 1b). The developer has high confidence in the quality of this code fragment because it was created by the JDT developers themselves. 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 integrated within his own code.

The developer considers refactoring the `computeMethodSignature(...)` method via "pull-up method", 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 1b 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 1b is identical to his own `StringBuffer` variable declaration statement on line 14 in Figure 1a. 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 1b, comparing it with his own usage of `signature.append(...)` on line 15 in Figure 1a; 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 1b corresponds with his `List` variable declaration statement on line 17 in Figure 1a. 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 1b corresponds with his own `for`-loop expression on line 18 in Figure 1a. 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 1b corresponds with the return statement on line 22 in Figure 1a. Statements from line 19 to line 26 in Figure 1b 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

[1] http://www.eclipse.org/jdt

```
1  public class MethodSignVisitor extends ASTVisitor{
2
3    LinkedList<String> strList;
4
5    public MethodSignVisitor(){
6      strList = new LinkedList<String>();
7    }
8    public boolean visit(MethodDeclaration node) {
9          strList.add( this.getMethodSign(node) );
10         return super.visit( node );
11   }
12
13   public String getMethodSign(MethodDeclaration
         method){
14     StringBuffer signature= new StringBuffer();
15     signature.append( getParent(method) );
16
17     List parameters = method.parameters();
18     for (Iterator iter = parameters.iterator();
           iter.hasNext();) {
19       // TODO: Complete this body
20     }
21
22     return signature.toString();
23   }
24 }
```

(a) Developer's target context (paste seed: `getMethodSign(...)`)

```
1  public class BreakpointMethodLocator extends
       ASTVisitor {
2    ...
3    private String computeMethodSignature(
         MethodDeclaration node) {
4      if (node.getExtraDimensions() != 0 ||
5          Modifier.isAbstract(node.getModifiers() ) {
6        return null;
7      }
8      StringBuffer signature= new StringBuffer();
9      signature.append('(');
10     List parameters = node.parameters();
11     for (Iterator iter = parameters.iterator(); iter
           .hasNext();) {
12       Type type = ((SingleVariableDeclaration) iter.
             next()).getType();
13       if (type instanceof PrimitiveType) {
14         appendTypeLetter(signature, (PrimitiveType)
               type);
15       } else {
16         return null;
17       }
18     }
19     signature.append(')');
20     Type returnType;
21     returnType= node.getReturnType2();
22     if (returnType instanceof PrimitiveType) {
23       appendTypeLetter(signature, (PrimitiveType)
             returnType);
24     } else {
25       return null;
26     }
27     return signature.toString();
28   }
29
30   private void appendTypeLetter(StringBuffer
         signature, PrimitiveType type) {
31     ...
32   }
33   ...
34 }
```

(b) Original context, `org.eclipse.jdt.internal.debug.ui.actions.BreakpointMethodLocator` (copy seed: `computeMethodSignature(...)`)

Figure 1: The source code involved in the sample scenario: the method `computeMethodSignature(...)` on the right is to be reused in the class on the left, with the method `getMethodSign(...)` serving as the paste seed.

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. Many 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.

3. RELATED WORK

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 [17] 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 [22].

Reuse tasks can be divided into two phases: location and integration. Several approaches focus on identifying source code use examples. Basili *et al.* [2] 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 [18] that applies data mining association rules to determine reuse patterns. Xie and Pie [24] 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 [11]; 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.

Relatively little work has considered the integration phase. Notable exceptions are approaches requiring formal specifications to perform the integration phase (e.g., [8, 25]); 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 [9] allows developers to plan medium- to large-scale reuse tasks at the integration phase, and Procrustes [10]

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.

Duala-Ekoko and Robillard [6] describe an approach for tracking evolving code clones that are mapped across a system, with prototype support for their simultaneous editing. CReN [12] 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.

Jackson and Ladd [13] 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 [1] 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 [5] 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.

4. THE JIGSAW TOOL

Jigsaw is a plug-in to the Eclipse[2] 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).

Jigsaw performs generalization over pairs of ASTs for the integration of small-scale reuse tasks.[3] Jigsaw acts on the copy & paste seeds, as outlined by the algorithm INTEGRATE: (1) input into the algorithm are the ASTs for the seeds and their surrounding classes, created via Eclipse's standard functionality; (2) generalization 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 AST that we call a *correspondence AST* (CAST); (4) evaluation of candidate correspondences to obtain the "best" correspondences (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 best-fit generalization (lines 4–6) to propose an integration solution for the developer's system (see Section 4.5).

[2]http://www.eclipse.org, v3.3

[3]Jigsaw performs higher-order anti-unification modulo theories (see Section 6.3). We utilize terms such as "generalization" and "correspondence" here, for simplicity.

INTEGRATE($copy, paste$)

1 $(cast[copy], cast[paste]) \leftarrow$ JIGSAW-GENERALIZE($copy, paste$)
2 $cast \leftarrow$ GENERALIZATION-OF-BEST-FIT($cast$)
3 CREATE-RELEVANCE-LINKS($cast[copy]$)
4 **for** each $k_a \in node[cast[copy]]$
5 **do if** $corresp[k_a] = \{\}$
6 **then** INSERT-MISSING-NODE($k_a, cast[paste]$)

To start the generalization process, Jigsaw considers the nature of the seeds and their surrounding contexts, via the JIGSAW-GENERALIZE algorithm. If the paste seed is a method, the copy & paste seed methods 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. Each field declaration node in the paste class AST is compared to every field declaration node in the copy class AST (lines 3–5). Each method declaration node in the paste class AST is compared with every method declaration node in the copy class AST (lines 6–8).

JIGSAW-GENERALIZE($copy, paste$)

1 **if** $seed[paste]$ instanceof `MethodDeclaration`
2 **then** GENERALIZE($seed[copy], seed[paste]$)
3 **for** each $k_{copy} \in fields[copy]$
4 **do for** each $k_{paste} \in fields[paste]$
5 **do** GENERALIZE(k_{copy}, k_{paste})
6 **for** each $k_{copy} \in methods[copy] \setminus \{seed[copy]\}$
7 **do for** each $k_{paste} \in methods[paste] \setminus \{seed[paste]\}$
8 **do** GENERALIZE(k_{copy}, k_{paste})

4.1 Running example

We introduce an example from the Motivational scenario (see Section 2) to assist in explaining the generalization process over two ASTs. We limit the example to the method parameter subtrees from the copy & paste method seeds shown in Figure 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.

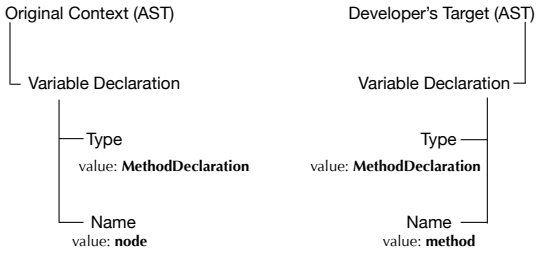

Figure 2: The ASTs of the "copy & paste" seed method parameters `MethodDeclaration method` and `MethodDeclaration node` from the Motivational scenario (see Section2).

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 generalization process; the similarity measure takes values between zero and one. GENERALIZE computes the similarity $sim[a,b]$ of the CAST nodes a and b; if this is greater

than 0, the presence of a candidate correspondence g between a and b is recorded in the CAST.

GENERALIZE(a,b)

```
1  Comparator ← LOOKUP-COMPARATOR(type[a], type[b])
2  if Comparator = NIL
3     then g[a,b] ← NIL
4          sim[a,b] ← 0
5     else ↑Comparator(a,b)
```

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; some of these heuristics leverage context-of-use information to improve the accuracy of the similarity measure. For example, let us supply the GENERALIZE algorithm with the method parameters from the seed methods from Section 4.1. Figure 3 shows how GENERALIZE is applied at each level of the AST through the use of the *Comparator*.

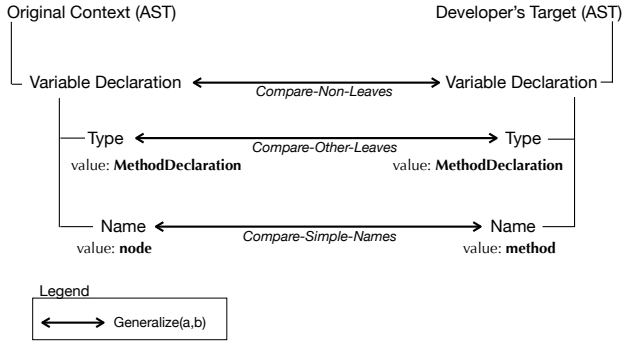

Figure 3: The generalization algorithms presented in Section 4.2 applied against the method parameters of the copy & paste method seeds (see Section 4.1).

The algorithm LOOKUP-COMPARATOR selects the most appropriate comparison algorithm for nodes of type T_1 and T_2. 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 `VariableDeclaration`s (i.e., COMPARE-NON-LEAVES), `Type` (i.e., COMPARE-OTHER-LEAVES), and `Name` (i.e., COMPARE-SIMPLE-NAME) which are shown in Figure 3. Calls to the algorithm GENERALIZE that did not return a *Comparator* (e.g., comparison of `Type` and `Name`) are not shown in Figure 3.

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 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.

COMPARE-SIMPLE-NAMES(a,b)

```
1  s_a ← string[a]
2  s_b ← string[b]
3  sim[a,b] ← w_N ×
     size[LONGEST-COMMON-SUBSTRING(s_a, s_b)] / max{s_a, s_b}
4  if sim[a,b] > 0
5     then g[a,b] ← MAKE-CORRESP-ANNOTATION()
```

For example, let us supply COMPARE-SIMPLE-NAMES with the `Name` nodes containing the values `node` and `method` from the ASTs in Figure 3. LONGEST-COMMON-SUBSTRING(`node`,`method`) will return 2 and the *max* of the two names is 6, thus resulting in a similarity of $\frac{1}{3}$.

Other leaf nodes are compared via the 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 of 1 returns the best results, again through informal experimentation. Otherwise, their similarity is defined as 0 (line 4). For example, let us supply COMPARE-OTHER-LEAVES with the `Type` nodes from the ASTs in Figure 3. The two types are equal, thus resulting in a similarity of 1.

COMPARE-OTHER-LEAVES(a,b)

```
1  if a = b
2     then sim[a,b] ← w_L
3          g[a,b] ← MAKE-CORRESP-ANNOTATION()
4     else sim[a,b] ← 0
5          g[a,b] ← NIL
```

The comparison of non-leaf nodes proceeds recursively, via the COMPARE-NON-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 set of these maxima are summed 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 COMPARE-NON-LEAVES with the `VariableDeclaration` nodes from the ASTs in Figure 3. The children of the `VariableDeclaration`s are `Type` and `Name` for each. The call to GENERALIZE(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 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 $\frac{2}{3}$.

COMPARE-NON-LEAVES(a,b)

```
1  best ← 0
2  for each k_a ∈ children[a]
3       do max ← 0
4          for each k_b ∈ children[b]
5               do GENERALIZE(k_a, k_b)
6                  max ← max(max, sim[k_a, k_b])
7          best ← best + max
8  sim[a,b] ← min(best, |a|, |b|) / max(|a|, |b|)
9  ▷ Each concrete implementation of COMPARE-NON-LEAVES
   ▷ can perform semantically-based heuristic adjustments here
10 if sim[a,b] > 0
11    then g[a,b] ← MAKE-CORRESP-ANNOTATION()
```

For different concrete implementations of this algorithm, semantically-based heuristics are applied to adjust the similarity measure to compare nodes that would not otherwise be comparable. Jigsaw does this by abstracting the common parts of the two structures that can be compared. Specifically, Jigsaw compares nodes that fall into one of these categories: variable declarations, conditionals, loops, statements, expressions, method declarations, type declarations, type references, exception handling, and import declarations.

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 as the type is only partially corresponding. For example, the formal parameter `MethodDeclaration method` in Figure 3 will also be compared with the variable declarations contained within the method on Lines 8, 10, 12, and 20 in Figure 1b.

The conditionals comparator allows comparison of `if`- and `switch`-statements. The test condition of the `if`-statement is compared against both the test condition of the `switch`-statement and each `case`'s expression. Despite the fact that the semantic similarity of these expressions will tend to be low to non-existent, these comparisons aid to identify of partial correspondences that are flagged for the developer's attention.

```
Iterator iter = parameters.iterator();
while(iter.hasNext()) { }
```

Figure 4: This `while`-loop maintains a similar contextual meaning to the `for`-loop on line 18 in Figure 1a.

The loop comparator allows comparison of enhanced-`for`-, `for`-, `while`- and `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 `for`-loop on line 18 in Figure 1a with the `while`-loop from Figure 4. The type `Iterator` and halting condition `iter.hasNext()` are identical. Tracing the use of `iter` in each seed, we identify that the halting condition for each loop uses `iter` in an identical manner. In terms of the generalization process outlined in the algorithms above, the tracing of `iter` occurs in the COMPARE-NON-LEAVES algorithm at *children*$[a]$ on Line 2. The type of `iter` is `Iterator` in both cases that is most recently defined through a call to `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.

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 Generalization of best fit

A CAST constructed as described in the previous section still represents many possible generalizations—too many to be practical for the user. To overcome this, we create a *generalization of best fit* via thresholding and, if needed, the developer's input to resolve each node's list of correspondences to select the strongest. The GENERALIZATION-OF-BEST-FIT algorithm (called by INTEGRATE algorithm on line 3) shows how the decisions made by Jigsaw or by the developer are cascaded to other correspondences 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).

GENERALIZATION-OF-BEST-FIT(a)

```
1  for each k ∈ children[a]
2      do if |corresp[k]| > 0
3           then INCLUSION-THRESHOLD(k)
4         if |corresp[k]| ≥ 2
5           then DIFFERENCE-THRESHOLD(k)
6         if |corresp[k]| ≥ 2
7           then PROMPT-DEVELOPER-INPUT(corresp[k])
```

To reduce the number of decisions that the developer must make, we apply two thresholds over the CAST to filter correspondences (lines 2–5). Nodes that do not have correspondences have no effect on the generalization and thus are ignored. If the number of correspondences for a CAST node is still two or more after the thresholds have been applied, the developer is asked to supply input (lines 6 and 7).

Inclusion threshold. If the similarity of a correspondence is below the inclusion threshold value, the correspondence is removed via the INCLUSION-THRESHOLD algorithm. The algorithm traverses every candidate correspondence and removes those that are below the threshold value (lines 2–3). 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.

INCLUSION-THRESHOLD(a)

```
1  for each k ∈ corresp[a]
2      do if sim[k] ≤ inclusion-threshold
3           then corresp[a] ← corresp[a] \ {k}
```

Difference threshold. Once the INCLUSION-THRESHOLD has completed traversing the CAST and if the CAST node contains more than one correspondence, the DIFFERENCE-THRESHOLD algorithm is applied. The algorithm is used to remove correspondences (line 4) if their similarity is not sufficiently close to that of the correspondence (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.

DIFFERENCE-THRESHOLD(a)

```
1  best ← max(corresp[a])
2  for each k ∈ corresp[a] \ {best}
3      do if sim[best] − sim[k] > difference-threshold
4           then corresp[a] ← corresp[a] \ {k}
```

Developer input. If more than one correspondence still remains in a node, the developer is prompted with the *conflict resolution view* (see Figure 5) to decide which correspondence should be selected. In this view, the developer is presented with a node from their source (Figure 5a), its context of use in their code (Figure 5b), the set of potentially corresponding nodes plus the decision "Do not correspond" (Figure 5c), and the source context of the currently selected corresponding node (Figure 5d). The decision "Do not correspond" gives the developer the ability to decide that none of the available candidates corresponds with their node. 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.

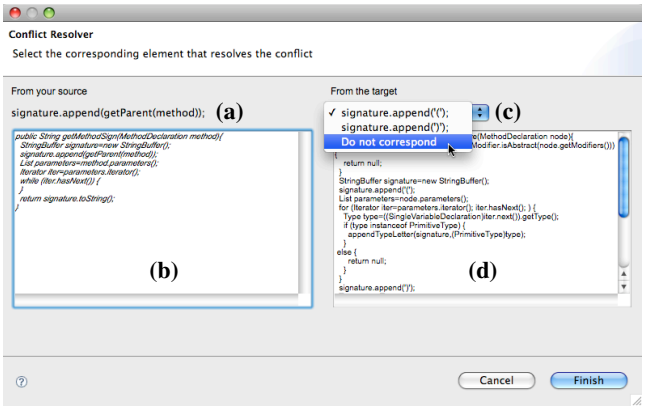

Figure 5: The conflict resolution view requests developer input to select a single correspondence.

For example, the `signature.append(...)` method call on line 15 in Figure 1a has a set of corresponding nodes, including a call to the `signature.append(...)` method on lines 9 and 19 in Figure 1b, that pass the inclusion threshold and are within the difference threshold. The developer is prompted for input (Figure 5) 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 an 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[4]; 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 2 is referenced on lines 4, 5, 10, and 21 in Figure 1b. For each of the references, relevance links will be created, as shown in Figure 6.

CREATE-RELEVANCE-LINKS($copy$)

```
1  for each k_a ∈ names[copy]
2      do for each k_b ∈ names[copy] \ {k_a}
3          do if binding[k_a] ≡ binding[k_b]
4              then rlink[k_a] ← rlink[k_a] ∪ {k_b}
5                   rlink[k_b] ← rlink[k_b] ∪ {k_a}
```

[4]Bindings are computed via Eclipse's Java Development Tools framework.

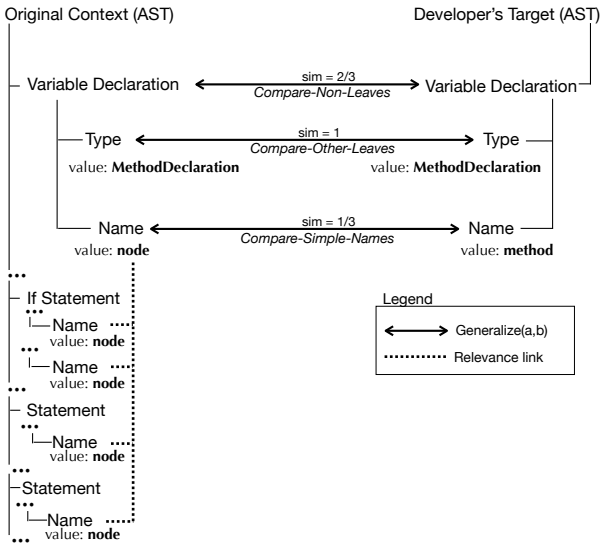

Figure 6: The CREATE-RELEVANCE-LINKS algorithm applied to the CAST of the method parameters of the copy & paste method seeds (see Section 4.1).

4.5 Integration

Once the CAST has been transformed into a representation of the generalization 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 relevance link to find the copy's context node that occurs in the generalization. This context node establishes the location and form of the "hole" into which the copy method's node should be inserted. The INSERT-MISSING-NODE algorithm performs the insertion.

INSERT-MISSING-NODE(a,b)

```
1  copy ← COPY-NODE(a)
2  ADD-NODE-TO-TREE(copy, b)
3  if rlink[a] ≠ NIL
4      then corresp[a] ← corresp[rlink[a]]
5           if corresp[a] = {}
6               then loc ← FIND-LOCATION(rlink[a], copy)
7                    INSERT-MISSING-NODE(rlink[a], loc)
8               else UPDATE(copy, corresp[a])
9  for each k ∈ children[a]
10     do INSERT-MISSING-NODE(k, copy)
```

Jigsaw constructs a new node (line 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 a correspondence (line 5), we update the name binding of the new node to that of the 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).

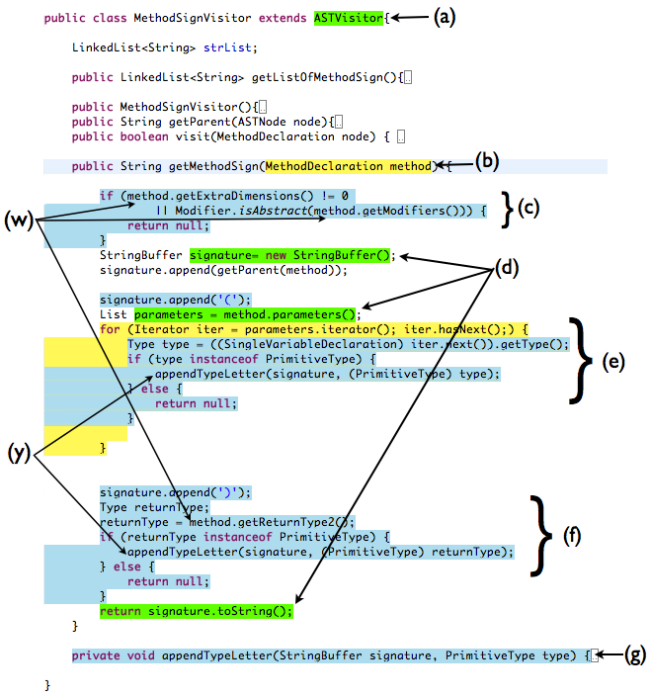

Figure 7: Jigsaw's output based on the motivational scenario of Section 2.

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 correspondences (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 1b has relevance links to the statements on lines 4, 5, and 21 that do not have correspondences; however, the declaration of `node` corresponds with the declaration of `method` on line 13 in Figure 1a (described in Section 4.4). Therefore, Jigsaw will copy lines 4 and 21 from Figure 1b and insert them into Figure 1a, replacing the name "`node`" with the name "`method`" (the result can be seen in Figure 7w). (2) In the copy context (Figure 1b), 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 correspondences. The `appendTypeLetter(...)` method calls and declaration are thus copied into the paste context (Figures 7y and 7g).

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 7). 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 7) 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 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(...)`.

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 [21]. The CCFinder[5] clone detection tool [14], 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 1.

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 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 8. 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

[5]Version 20071121 with default settings.

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

Table 1: Method pairs used as test cases; all are contained in the `org.eclipse.jdt` package.

```
1  public String toString() {
2    final StringBuilder buffer = new StringBuilder();
3    final IMethodBinding methodBinding =
         getDeclarationBinding();
4    final ITypeBinding[] typeParams = methodBinding.
         getTypeParameters();
5    if(typeParams != null && typeParams.length > 0){
6      buffer.append('<');
7      for(int i=0; i<typeParams.length; i++){
8        if( i != 0 ) buffer.append(``, '');
9        buffer.append(typeParams[i]);
10     }
11     buffer.append('>');
12   }
13
14   if(methodBinding.getReturnType() != null)
15     buffer.append(methodBinding.getReturnType().
           getName());
16
17   buffer.append(methodBinding.getName());
18   buffer.append('(');
19   int i = 0;
20   for(ParameterDeclaration param : getParameters()){
21   if(i++ != 0) buffer.append(``, '');
22     buffer.append(param);
23   }
24   buffer.append(')');
25   return buffer.toString();
26 }
```

Figure 8: Paste seed from test case 1 (`MethodDeclaration-Impl.toString()`).

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.[6] The results of this analysis are shown in Table 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 `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 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.

[6]The workstation evaluating the Jigsaw plug-in was a Macbook 2GHz Core 2 Duo with 2GB RAM.

Test case	Conflict resolution (#)	Correspondence		Integration		Time (seconds)
		Correct (%)	Wrong context	Correct (LOC) (%)	Incorrect (LOC)	
1	3	5 (83.3)	1	8 (88.9)	1	1.0†
2	0	1 (100)	0	11 (100)	0	0.1
3	0	3 (100)	0	4 (100)	0	2.7
4	0	3 (75)	1	32 (97)	1	5.2
5	1	5 (100)	0	11 (100)	0	0.7†
6	0	4 (100)	0	5 (100)	0	0.9
7	1	3 (100)	0	5 (100)	0	0.7†
7*	1	2 (100)	0	21 (100)	0	0.7†
8	0	3 (100)	0	7 (100)	0	0.5
9	0	5 (100)	0	7 (100)	0	1.0
10	6	15 (92.8)	1	12 (85.7)	2	1.1†
11	0	4 (100)	0	5 (100)	0	1.2
12	0	2 (100)	0	22 (100)	0	0.8
13	0	8 (100)	0	4 (100)	0	1.1
14	0	5 (100)	0	19 (100)	0	1.4
15	0	8 (100)	0	6 (75)	2	0.5

Table 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.

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 succeeded 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 insertion.

The use of multiple generalizations and a similarity measure that takes into account semantic knowledge for determining correspondence eliminated the mismatch and conceptual disconnect errors that we reported in our previous application of simple generalization [5]. The time cost accrued to perform the generalization and the integration of the reuse source in the developer's target was inconsequential.

In the four cases 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 need 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 skilled 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 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.

6. DISCUSSION

In this section, we discuss the validity of our evaluation and case studies, a number of remaining issues with Jigsaw, theoretical foundations and potential extensions.

6.1 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 [21].

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.

6.2 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.

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.

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.

6.3 Theoretical foundations

The method employed by Jigsaw is based on the concept of anti-unification [19]. More precisely, the need to deal with the fact that the sequence of the field and method declarations is of no consequence requires that we include equational theories in the anti-unification process. In addition to the declaration part being associative and commutative, we need other theories expressing semantics about loops and other control structures. In the future, we might include additional semantical knowledge. Anti-unification modulo theories (also called E-generalization) is already undecidable in general [4]. But we are, in principle, looking at an even more difficult problem, namely the fact that we need to abstract from method names (as well as names in general), which means that the general problem that was our starting point is higher-order anti-unification modulo theories.

Fortunately, the intended usage of Jigsaw—suggesting how to modify the source code of the copy method seed to fit into the holes in the paste seed method—provides us with obvious heuristics to choose among the many possible anti-unification abstractions of the paste and copy seeds. We also deliberately see Jigsaw as a semi-automated tool, that is primarily intended to assist the developer's task: determining a perfect correspondence would be nice if it were possible; in the absence of such a possibility, the developer is best assisted by having straightforward cases handled automatically and complex cases flagged for their attention.

We intend to pursue a formal description of the model used by Jigsaw, so that other applications and variations can be more easily instantiated in future.

6.4 Future work

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 in-

correct 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.* [23]. 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., [2, 18, 11, 24]) of software reuse could be improved by Jigsaw's underlying framework. An example is the Strathcona example recommender [11]. 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.

7. 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 generalization, 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 generalizing code in this manner is known to be undecidable in general; the goal-orientation of our tool allowed us to approximate the needed generalization 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 generalization 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 on the most difficult issues in a small-scale reuse task.

8. ACKNOWLEDGMENTS

Our thanks to the other members of the Laboratory for Software Modification Research for their on-going feedback and our industrial participants for their time and effort. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada and in part by IBM Canada.

9. REFERENCES

[1] T. Apiwattanapong, A. Orso, and M. J. Harrold. A differencing algorithm for object-oriented programs. In *Proc. Int'l Conf. Automated Softw. Eng.*, pp. 2–13, 2004.

[2] V. R. Basili, G. Caldiera, and G. Cantone. A reference architecture for the component factory. *ACM Trans. Softw. Eng. Methodol.*, 1(1):53–80, 1992.

[3] T. J. Biggerstaff. The library scaling problem and the limits of concrete component reuse. In *Proc. Int'l Conf. Softw. Reuse*, pp. 102–109, 1994.

[4] J. Burghardt. E-generalization using grammars. *Artif. Intell. J.*, 165(1):1–35, 2005.

[5] R. Cottrell, J. J. C. Chang, R. J. Walker, and J. Denzinger. Determining detailed structural correspondence for generalization tasks. In *Proc. Joint Europ. Softw. Eng. Conf. and ACM SIGSOFT Int'l Symp. Foundations Softw. Eng.*, pp. 165–174, 2007.

[6] E. Duala-Ekoko and M. P. Robillard. Tracking code clones in evolving software. In *Proc. Int'l Conf. Softw. Eng.*, pp. 158–167, 2007.

[7] W. B. Frakes and K. Kang. Software reuse research: Status and future. *IEEE Trans. Software Eng.*, 31(7):529–536, 2005.

[8] M. G. Gouda and T. Herman. Adaptive programming. *IEEE Trans. Softw. Eng.*, 17(9):911–921, 1991.

[9] R. Holmes and R. J. Walker. Supporting the investigation and planning of pragmatic reuse tasks. In *Proc. Int'l Conf. Softw. Eng.*, pp. 447–457, 2007.

[10] R. Holmes and R. J. Walker. Lightweight, semi-automated enactment of pragmatic-reuse plans. In *Proc. Int'l Conf. Softw. Reuse*, pp. 330–342, 2008.

[11] R. Holmes, R. J. Walker, and G. C. Murphy. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Trans. Softw. Eng.*, 32(12):952–970, 2006.

[12] P. Jablonski and D. Hou. CReN: A tool for tracking copy-and-paste code clones and renaming identifiers consistently in the IDE. In *Proc. Eclipse Technology Exchange*, pp. 16–20, 2007.

[13] D. Jackson and D. A. Ladd. Semantic diff: A tool for summarizing the effects of modifications. In *Proc. Int'l Conf. Softw. Maintenance*, pp. 243–252, 1994.

[14] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, 2002.

[15] C. Kapser and M. W. Godfrey. 'Cloning considered harmful' considered harmful. In *Proc. Working Conf. Reverse Eng.*, pp. 81–90, 2006.

[16] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *Proc. Joint Europ. Conf. Softw. Eng. and ACM SIGSOFT Int'l Symp. Foundations Softw. Eng.*, pp. 187–196, 2005.

[17] C. W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, 1992.

[18] A. Michail. Code web: Data mining library reuse patterns. In *Proc. Int'l Conf. Softw. Eng.*, pp. 827–828, 2001.

[19] G. D. Plotkin. A note on inductive generalization. *Machine Intelligence*, 5:153–163, 1970.

[20] M. B. Rosson and J. M. Carroll. The reuse of uses in Smalltalk programming. *ACM Trans. Computer–Human Interaction*, 3(3):219–253, 1996.

[21] F. Van Rysselberghe and S. Demeyer. Reconstruction of Successful Software Evolution Using Clone Detection. *Proc. Int'l Wkshp. Principles of Softw. Evolution*, pp. 126–130, 2003.

[22] R. W. Selby. Enabling reuse-based software development of large-scale systems. *IEEE Trans. Softw. Eng.*, 31(6):495–510, 2005.

[23] F. Tip, A. Kiezun, and D. Bäumer. Refactoring for generalization using type constraints. In *Proc. ACM SIGPLAN Conf. Object-Oriented Progr. Syst. Lang. Appl.*, pp. 13–26, 2003.

[24] T. Xie and J. Pei. MAPO: Mining API usages from open source repositories. In *Proc. Int'l Wkshp. Mining Softw. Repositories*, pp. 54–57, 2006.

[25] D. M. Yellin and R. E. Strom. Protocol specifications and component adaptors. *ACM Trans. Program. Lang. Syst.*, 19(2):292–333, 1997.