Unique Renaming of Java Using Source Transformation

Xinping Guo
James R. Cordy
Thomas R. Dean

School of Computing, Queen's University
Kingston, Ontario, Canada K7L 3N6
{guo,cordy,dean}@cs.queensu.ca

Abstract

This paper presents a flexible way in which a deisgn model extracted from Java programs can remain unified with the source code. Each entity declaration and reference in the Java program is assigned a globally unique identifier (UID) based on its declaration scope and file which serves as a key by which its original declaration and other references can be found, and more importantly, by which information about the entity can be stored or retrieved from the design database. The resulting uniquely renamed source code makes it convenient and efficient to do further business logic and technical analysis that crosses the boundary between source code and the design database.

The UIDs are attached to the entity references in the source code using XML markup, so that both the UID and the original source text of the declaration or reference are available in the renamed source program. While is possible to generate the unique names in an ad hoc manner, we show how to generate these using a combination of source transformations and design database inferences. This ensures that the notion of UID is consistent and well defined.

1. Introduction

As software comprehension technology matures and comes into practice in system transformation and migration tasks, attachment to source becomes increasingly important. While embedded AST and lexical coordinate methods serve well in attaching analysis results to static source, they fare less well when used in the context of source transformation and automated reprogramming tasks.

Unique naming [9] uses a different approach. Unique naming provides a unifrom position-independent means of associating the declaring and referencing instances of an entity in the source code with the corresponding entity in the design database. Using unique naming, all instances referring to an entity in both source and graph use the same unique name, which serves as a kind of key linking the one to the other.

This representation has two quite distinct advantages - first, because analysis artifacts are represented with both source and graph representations, the design graph need not be constrained to carry source information, and the source need not be restructured to match the design representation, making it easier to analyze and transform each. Secondly, because source and design are implicitly linked to one another by the entity unique names, analysis and transformation of source and analysis and transformation of design can each be carried out independently without losing the connection to the other. Tasks more appropriate to source can be carried out on source, and tasks more appropriate to design can be carried out on the design database. In either case, the changed result of tasks carried out on one are automatically attached to the other by the unique names of the entities involved.

In order to be effective, unique names must provide entity identifiers that are globally unique across entire systems, while at the same time easy to derive locally so that source files can be uniquely named on a file by file basis. One way to do this is to derive the unique identifier (UID) of each entity from the source file and scope context of its defining declaration, in the style of internet URL's. This approach has the advantage that the containment relations for the entity are encoded directly in its UID, effectively making the containment relationship redundant in the design database and allowing for scope-dependent analyses and transformations to be done directly in the uniquely named source code without reference to the design graph.

By encoding the UID in source declarations and references using markup rather than replacement, we preserve the original source text of each reference and declaration in the uniquely named source while attaching each to the design database. Unique naming using markup has been used for COBOL, RPG and PL/I systems in LS/2000 [9] and has been extended to complex analysis of maintenance hotspots in these languages using HSML [31].

While unique naming has worked well for analysis and transformation of these older procedural languages, legacy systems in object-oriented languages such as Java [27] are an increasing concern. In this paper we extend the scope-based unique naming of these older languages to handle the more complex scope and inheritance rules of modern object-oriented languages, and show how object-oriented linking can be reflected directly into source using unique name resolution. We introduce a method for resolution of unique names in the presence of class instances and inheritance, and extend the notion to handle shared object-oriented class libraries.

We use XML [1] tags to as the markup notation to add unique naming tags to Java source. XML is already widely used

in reverse engineering systems to represent both syntactic analysis information and software designs [2, 3, 4, 5, 6, 14]. XML tags provide a flexible way to represent many kinds of information, such as syntax and semantic analysis information, design graphs, source elisions, and so on, which can be built on our unique renaming.

2. Unique Identifiers

Java program comprehension and Java design recovery are currently hot research topics. The use of name spaces in Java programs makes these activities more challenging when working with source artifacts. For example, it is possible that all of the following names in a Java program could be the same:

- Class, constructor and field names in a class.
- Two class names in different packages.
- Overloaded method names.
- Local variables in different methods.
- Local / parameter variables and class / instance variables.

One way to distinguish between these entities is by using scope rules to give each declared name a unique identifier (UID) [9]. Sample source code and generated UIDs are shown in Figure 1. We use this sample throughout the paper to demonstrate the process of generating and manipulating UIDs.

In our schema, the UID of an entity is composed "inside out" based on the original name of the entity, along with information encoding the entity's scope and location in the code, in the form:

"entity_name enclosing_class_and_interface_names package_name file_name"

Using this schema, in the example of Figure 1 we can see:

- The global and local variables named *x* are assigned the different UIDs "*x **Ex** bar foo Ex.java*" and "*x **main** Ex bar foo Ex.java*" respectively,
- The local variables named *y* in the two methods are assigned the different UIDs, "*y **x** Ex bar foo Ex.java*" and "*y **main** Ex bar foo Ex.java*".

Using this schema, two classes in different packages with same name would also have different UIDs because the package names are included in the UIDs, and so on. However, in the example we see that both the field and the method *x* have been assigned the same UID "*x Ex bar foo Ex.java*". The reason is that it is not necessary to distinguish these two, because in both source code and the design database it is unambiguous which is meant in every context. In source code, field references and method references are easily distinguished because method declaration and invocation always includes parentheses with zero or more arguments. In the design database, field and method entities are contextually distinguished by the database schema. This was a conscious decision in our model. If necessary, method and field UIDs can be distinguished simply by adding the word "method" or "field" in the UID strings as appropriate.

3. The Java Unique Renaming Process

Our unique renaming process involves assigning UIDs to all declared entities in the program. UIDs are assigned based on declaration context in the source code. However, references in the code to external and library entities must also be annotated.

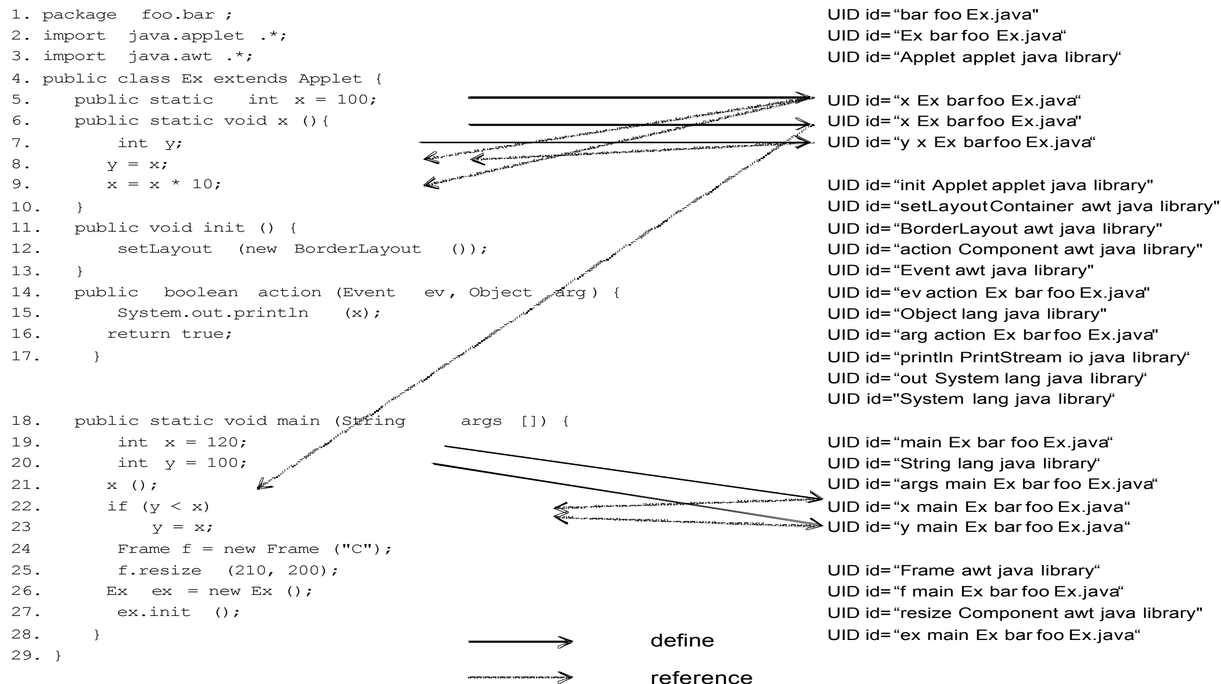

Figure 1. The use of Unique Identifiers in a Sample Java Program.

In addition, a local class can be declared within a block of Java code and a class may contain fields, constructors, methods, classes, interfaces or static/instance initializers. And to meet our schema, each UID must present the path to the entity layer by layer inside-out. These properties of the Java language make unique renaming an interesting and challenging task. Our technique is described in the following subsections.

There are eight steps in Java unique renaming process, described in detail in the eight following sections. The eight steps are as follows:

1. Uniquely rename declarations in each class file.
2. Uniquely rename direct references to declarations in each class file.
3. Uniquely rename the Java library interfaces (once for all programs).
4. Uniquely rename direct references to external and library entities in each class file.
5. Uniquely rename reference qualifications in each class file.
6. Extract a simple data design model from the renamed library and all program class files.
7. Analyze the data model to link all external reference unique names to their external target library and class file entities.
8. Implement the links in each class file to replace external reference unique names with the unique names they are linked to.

When the process is complete, every entity declaration and reference in every class file in the program will be globally uniquely renamed to refer directly to the UID of the referenced entity.

3.1 Step 1: Uniquely Rename Declarations

The goal of this first step is to assign each declared name in each class file a UID in the source code. The entities to be annotated include package, class, interface, variable, constructor, and method names. UIDs are created layer by layer from the inside-out. The Java source code is parsed by our TXL program according to a standard Java base grammar [23]. The resulting parse tree encodes the scope structure of the program. By traversing the parse tree from the bottom up, the scope structure of the program is inferred and stored in generated UIDs.

Adding file and package names is trivial; however, adding other information is tricky, especially for inner classes. The use of recursive transformation rules is a good solution for handling these cases. Using the source file name, we begin by annotating all declarations with XML tags containing partial UIDs of the form:

<UID id="file_name"> declared_name </UID>

We then incrementally add each level of scope information to the UIDs in the XML tags one level at a time by processing each scope of the program from the outside in. The final result UID encodes the file name, declared name and the names of all enclosing scopes in inside-out order:

*<UID id="declared_name inner_scope_name …
outer_scope_name file_name"> declared_name </UID>*

Figure 2 shows the annotated source code from this first step for the sample program in Figure 1. When we are done, the UID for each declared entity represents all of its scope information. For example, the UID *"y x Ex bar foo Ex.java"* means that local variable *y* is located in method *x*, method *x* is in class *Ex*, class *Ex* is in package *foo.bar* and the source class file name is *Ex.java*. As shown in Figure 2, UIDs are assigned for all declared names, including:

- Package *foo.bar*.
  <UID id="bar foo Ex.java"> foo.bar</UID>
- Class *Ex*.
  <UID id="Ex bar foo Ex.java"> Ex</UID>
- Field *x*.
  <UID id="x Ex bar foo Ex.java"> x</UID>
- Methods *x, init, action* and *main*.
  <UID id="init Ex bar foo Ex.java"> init</UID>
- Parameters *ev, arg and args*.
  <UID id="ev action Ex bar foo Ex.java">ev</UID>
- Local variables *y* in *x(), y* in *main(), x, f* and *ex*.
  <UID id="f main Ex bar foo Ex.java"> f</UID>

All declared names inside class *Ex* contain "*… Ex bar foo Ex.java*" in their UIDs. All declared names nested in a method contain the method name as well. For example, in the *main()* method, all UIDs of declarations end with "*… main Ex bar foo Ex.java*". This property makes our UIDs a rich source of scoping information which can be exploited to optimize source analysis and transformation tasks by avoiding reference to the design database to look up structural information such as containment relationships.

3.2 Step 2: Uniquely Rename Direct References

Once all declarations have been annotated with their unique names as shown in Figure 2, we must annotate the direct references to these declared names in each class file. By direct references we mean the unqualified names on the left of reference expressions, for example, *f* in *f.resize()*. We will uniquely rename the qualifications themselves in a later stage of the process.

Most references are internal – references to declared names in the class file itself (e.g. variables *y* and *f*), but others are external references to entities in other library or class files (e.g. *Applet* and *Frame*). Binding references to declarations must of course follow the scope rules of the language, so the correct UID for any reference is the UID of the declaration of the entity to which the reference refers. For example, in Figure 1, the *x* in the statement *x=x*10* is a reference to the variable *x* declared in the statement *public static int x = 100*, so the UID for the reference is "*x Ex bar foo Ex.java*".

The steps in renaming of references mirror the steps in the scope rules of Java. We begin in this section with the simplest case, resolving direct references to already uniquely named declarations and annotating each with the UID of the

```
package <UID id="bar foo Ex.java">foo.bar</UID> ;
import java.applet.*;
import java.awt.*;
public class <UID id="Ex bar foo Ex.java">Ex</UID> extends Applet {
  public static int <UID id="x Ex bar foo Ex.java">x</UID> = 100;
  public static void<UID id="x Ex bar foo Ex.java">x</UID> () {
    int <UID id="y x Ex bar foo Ex.java">y</UID> ;
    y = x ;
    x = x * 10;
  }
 public void <UID id="init Ex bar foo Ex.java">init</UID> () {
    setLayout( new BorderLayout());
 }
 public boolean <UID id="action Ex bar foo Ex.java">action</UID> (Event<UID id="ev action Ex bar foo Ex.java">ev</UID> ,
    Object <UID id="arg action Ex bar foo Ex.java">arg</UID> ){
    System.Out.println (x);
    return true;
 }
 public static void <UID id="main Ex bar foo Ex.java">main</UID> (String
     <UID id="args main Ex bar foo Ex.java">args</UID> []){
    int <UID id="x main Ex bar foo Ex.java">x</UID> = 120;
    int <UID id="y main Ex bar foo Ex.java">y</UID> = 100;
    x();
    if (y < x)
      y = x;
    Frame <UID id="f main Ex bar foo Ex.java">f</UID> = new Frame("C");
    f.resize(210, 200);
    Ex <UID id="ex main Ex bar foo Ex.java">ex</UID> = new Ex();
    ex.init();
 }
}
```

Figure 2. Uniquely Rename Declarations.

declaration. Unique naming of references to libraries and external packages is handled in the following sections.

Finding and naming entity references within their declaration scope is not particularly difficult. However, some declarations may be masked in part by another declaration of the same name [13]. For example in Figure 1, there are two variables named *x*. The scope of the instance field *x* (*UID id="x Ex bar foo Ex.java"*) is the entire body of class *Ex*. Therefore inside method *x ()*, the reference to *x* refers to that global variable. However, in method *main ()*, there is a local variable *x* (*UID id="x* ***main*** *Ex bar foo Ex.java"*) which masks the declaration of the global variable, and therefore the reference to *x* in *main ()* refers to the local declaration, i.e. (*UID id="x* ***main*** *Ex bar foo Ex.java"*).

We tag each direct reference with the UID of the declared entity it refers to, if any. If none is found, the reference is assumed to be external and is handled in the step four.

When this step is complete, all declarations and their internal direct references in each class file have been annotated with UIDs as shown for our sample code example in Figure 3.

While direct references to *x*, class *Ex* and local variables *x, y, f* and *ex* have been uniquely renamed in the source of Figure 3, there still remain several unresolved references. These are of two kinds. First, there are references to external Java libraries:

- Class types: *Applet, System, Frame …*
- Inherited methods: *setLayout(), init() …*

Second, there are qualified references:

- Qualifiers: Entities following "." in qualified names: *f.resize(), ex.init() …*

We continue by addressing the first issue, references to external class files and Java libraries.

3.3 Step 3: Uniquely Rename Java Library Interfaces

In order to accurately uniquely rename external references to the library in Java source code, we must first address the question of uniquely renaming the Java library files themselves. Otherwise we cannot be sure that the UIDs used to refer to library entities are consistent across all Java programs in the system. We need only do library unique renaming once – we can store the results and use the same unique names to resolve external Java library references in all Java programs.

A difficulty with this approach is the fact that we do not have the sources for all of the Java libraries. As it turns out, this is not a serious problem – we can use the *javap* command to disassemble the Java class files of the library to source interfaces and rename those instead. The output of *javap* is an interface source file similar to the original library source code but without package name and implementation details. A small TXL source transformation is used to normalize these library interface files by extracting the package name and removing it from the names of inner classes and constructors. We then simply use the declaration and reference unique renaming transformations of Sections 3.1 and 3.2 to implement unique renaming of the library. In place of the source class file name, the special file name "*library*" is used in all library UIDs to ensure consistency among library files.

Unique renaming of references in library interface files is straightforward because all references in the library interfaces extracted by *javap* are fully qualified, and thus we can simply expand them directly. For example, the UID of the reference *java.awt.Panel.AccessibleAWTPanel* is *"AccessibleAWTPanel Panel awt java library"*.

Once renaming of the Java library is done once, it need never be done again and can be reused for renaming of all future programs.

```
package<UID id= "bar foo Ex.java">foo.bar</UID> ;
...
public class <UID id= "Ex bar foo Ex.java">Ex</UID> extends Applet{
  public static int <UID id="x Ex bar foo Ex.java">x</UID> = 100;
  public static void<UID id="x Ex barfoo Ex.java">x</UID> () {
    int <UID id= "y x Ex barfoo Ex.java">y</UID> ;
    <UID id= "y x Ex barfoo Ex.java">y</UID> = <UID id= "x Ex bar foo Ex.java">x</UID> ;
    <UID id= "x Ex barfoo Ex.java">x</UID> = <UID id= "x Ex barfoo Ex.java">x</UID> * 10;
  }
 ...
 public boolean <UID id="action Ex barfoo Ex.java ">action</UID> (Event<UID id="ev action Ex barfoo Ex.java">ev</UID> ,
    Object <UID id="arg action Ex barfoo Ex.java">arg</UID> ){
    System.Out.println( <UID id= "x Ex barfoo Ex.java">x</UID> );
    return true;
 }

 public static void <UID id= "main Ex bar foo Ex.java">main</UID> (String
    <UID id="args main Ex bar foo Ex.java">args</UID> [] ) {
    int <UID id= "x main Ex bar foo Ex.java">x</UID> = 120;
    int <UID id= "y main Ex bar foo Ex.java">y</UID> = 100;
    <UID id="x Ex bar foo Ex.java">x ()</UID> ;
    if ( <UID id="y main Ex barfoo Ex.java">y</UID> < <UID id="x main Ex barfoo Ex.java">x</UID> )
        <UID id="y main Ex bar foo Ex.java">y</UID> = <UID id= "x main Ex bar foo Ex.java">x</UID> ;
    Frame <UID id= "f main Ex barfoo Ex.java">f</UID> = new Frame("C");
    <UID id="f main Ex bar foo Ex.java">f</UID> . resize(210, 200);
    <UID id="Ex bar foo Ex.java">Ex</UID> <UID id= "ex main Ex barfoo Ex.java">ex</UID> =
      new <UID id="Ex bar foo Ex.java">Ex</UID> ();
    <UID id= "ex main Ex barfoo Ex.java">ex</UID> . init();
  }
}
```

Figure 3. Uniquely Rename Direct References.

3.4 Step 4: Uniquely Rename External Direct References

Once we have the uniquely renamed the libraries, we ready to uniquely rename the external references in any Java program. Once again, at this stage we only rename direct references – those unqualified names on the left of reference expressions, in each class file. Qualifications will be uniquely renamed in the next step. The result of these two steps on our sample program is shown in Figure 4.

We uniquely rename all references to types (class types and interface types) declared in the library. We do this by resolving the names used in all unrenamed direct references with the declarations in the uniquely renamed libraries created in step 3.

In the sample program of Figure 4, references to *Applet, BorderLayout, Event, Object, System, String* and *Frame* have all been uniquely renamed as library references. For example, the UID of *Applet* was found to be *"Applet applet java library*", denoting a reference to the *Applet* class of the Java library.

Once we have resolved all library references, the only remaining unresolved direct references must be to external user symbols such as inherited members of other class files. To complete the unique renaming of references, we assign each of these references a temporary UID as if they were declared in current class, for example, the reference to the inherited member *setLayout()* of our sample program is approximated as *"setLayout Ex bar foo Ex.java"*. Section 3.7 describes our strategy for the linking of these temporary UIDs to the actual external entities they refer to in other class files.

3.5 Step 5: Uniquely Rename Qualified References

Once unique names have been assigned to all external references, there remain no unresolved direct references in the program. At this point we refine the UIDs for qualified references (those using "."), e.g. *System.out.println(), f.resize()* and *ex.init()*. Previous steps have already uniquely renamed the direct (base) reference of every qualified name. For example,

<*UID id=*"*System lang java library*">System</*UID*>
.out.println ( )

In this step we expand these base references to create UIDs for the entire qualified references level by level. This is actually a very simple process. Since Java does not allow partial qualification, the UID for a reference *x.y* where *x* has UID *"x blat bar foo prog.java"* is always simply *"y x blat bar foo prog.java"*, that is, *y* followed by the UID for *x*.

By applying this algorithm at every level, we get fully renamed qualified names. For example, the uniquely named qualified expression *System.out.println()* shown above becomes:

<*UID id="println out System lang java library"*>
<*UID id="out System lang java library"*>
<*UID id="System lang java library"*>
System</*UID*>.out</*UID*>.println()</*UID*>

The result of the resolution of qualified names for our sample program is shown in Figure 4.

While this simple qualification naming algorithm works correctly for most internal qualified references, note that at this stage UIDs for qualified references are really an approximation. Firstly, qualified object references have been renamed as if the fields and methods were members of the object, when in fact they should be renamed to refer to the corresponding class member declaration. Secondly, external references are an approximation because we have not yet taken into account inheritances and overrides that may be present in the external class hierarchies.

```
package<UID id="bar foo Ex.java"> foo.bar</UID> ;
...
public class <UID id="Ex bar foo Ex.java">Ex</UID> extends <UID id="Applet applet java library">Applet</UID> {
 ...
 public void <UID id= "init Ex bar foo Ex.java">init</UID> () {
   <UID id="setLayout Ex bar foo Ex.java">setLayout( new <UID id= "BorderLayout awt java library">BorderLayout</UID>
   () ) </UID> ;
 }

 public boolean <UID id= "action Ex barfoo Ex.java">action</UID> (
    <UID id="Event awt java library">Event</UID>   <UID id="ev action Ex barfoo Ex.java">ev</UID> ,
     <UID id="Object lang java library">Object</UID>   <UID id="arg action Ex bar foo Ex.java"> arg</UID> ) {
    <UID id="println out System lang java library">  <UID id="out System lang java library">
      <UID id="System lang java library">System</UID> . Out</UID>
       . println( <UID id= "x Ex barfoo Ex.java">x</UID> ) </UID>;
    return true;
 }

public static void <UID id= "main Ex bar foo Ex.java">main</UID> (<UID id= "String lang java library">String</UID>
    <UID id="args main Ex bar foo Ex.java"> args</UID> [] ) {
   ...
   <UID id= "Frame awt java library">Frame</UID>  <UID id= "f main Ex bar foo Ex.java"> f </UID> =
     new <UID id="Frame awt java library">Frame</UID> ("C");
   <UID id= "resize f main Ex barfoo Ex.java">  <UID id="f main Ex barfoo Ex.java"> f</UID> . resize(210, 200)</UID>;
   <UID id= "Ex bar foo Ex.java">Ex</UID>   <UID id= "ex main Ex bar foo Ex.java">ex</UID>  =
    new <UID id="Ex bar foo Ex.java">Ex</UID> ();
   <UID id= "init ex main Ex barfoo Ex.java "> <UID id= "ex main Ex bar foo Ex.java">ex</UID> . init()</UID>;
 }
}
```

Figure 4. Uniquely Rename External and Qualified References.

For example, the UID *"println out System lang java library"* should really refer to *"println PrintStream io java library"*. In the next two sections, we address both these issues by implementing a link analysis of the entire renamed data model of all program class files and the library together.

3.6 Step 6: Extract the Data Design Model from Uniquely Renamed Source Files

At this point in the unique renaming process we have assigned unique names to every declaration and reference in all of the class files of the Java program. In the class files, only those references to apparently external, inherited entities have been assigned the correct UIDs. Other references (i.e. qualified and inherited members) are represented by internal approximate UIDs. The problem now is to link these approximate local UIDs to the real UID of the entity to which they refer. In order to do this, we use a static data design model to imitate the actions of a Java run-time linker.

We begin by extracting a database of data design facts from each of the uniquely renamed source files in the program. This database can be useful in many design analysis tasks, but in particular, we can use it to resolve the actual entity targets of our approximate external UIDs in section 3.7.

We use the design recovery technique described by Schneider et al [29] to infer and gather data design facts from our uniquely renamed source and library files. The method uses TXL rules to search for patterns in the source and annotate the source with design facts [8]. The output facts can be in the any format, for example Prolog, TA or RSF.

The following sections describe the facts inferred by our data design recovery for use in linking.

3.6.1. Accessibility Facts. Access control is the mechanism in Java that prevents the users of a package or class from depending on unnecessary details of the implementation of that package or class [13]. The access modifiers **public**, **protected** and **private** are used to specify access control, with default access defined by the Java language specification. We extract access facts for all classes, interfaces, fields, constructors and methods. For our example sample program, we get:

public ("Ex bar foo Ex.java")
public ("x Ex bar foo Ex.java")
public ("init Ex bar foo Ex.java")

3.6.2. Entity Facts. We extract entity facts from the renamed library files. These describe the defined packages, classes and interfaces in the library files. Example facts from the Java library are:

package ("applet java library")
class ("Applet applet java library")
class ("AccessibleApplet Applet applet java library")
package ("applet java library")
interface ("AppletContext applet java library")

3.6.3 Type Structure Facts. These describe the type relationships between entities including inheritance, abstraction, members, inner types and static members. We derive the following facts:

- *hasFieldType* – Represents the types of declared fields, e.g., *hasFieldType ("x Ex bar foo Ex.java", int)*
- *hasMethodType* – Represents the types of declared methods, e.g., *hasMethodType ("x Ex bar foo Ex.java", void)*
- *hasMemberForClass* – Represents the member relationship, e.g., *hasMemberForClass ("Ex bar foo Ex.java", "x Ex bar foo Ex.java")*

- *hasInnerType* – Represents the inner class relationship of classes and interfaces.
- *hasSuperType* – Represents the *extends* relationship, e.g.,
  hasSuperType ("Ex bar foo Ex.java",
  "Applet applet java library")
- *hasImplement* – Represents the *implements* relationship.
- *static* – Represents the *static* property, e.g.,
  static ("x Ex bar foo Ex.java")

3.6.4. Method Facts. Parameter, local variable and type facts are extracted at the level of methods and constructors. These facts encode not only implementation detail but also relationships with other classes and interfaces.

- *paramVar* – Method has parameter, *e.g*.,
  paramVar ("action Ex bar foo Ex.java",
  "ev action Ex bar foo Ex.java")
- *localVar* – Method has local variable, e.g.,
  localVar ("x Ex bar foo Ex.java",
  "y x Ex bar foo Ex.java")
- *hasVarType* – Method references type, e.g.,
  hasVarType ("action Ex bar foo Ex.java",
  "Event awt java library")
- *funRef* – Method references method, e.g.,,
  funRef ("init Ex bar foo Ex.java",
  "setLayout Ex bar foo Ex.java")
- *varRef* – Method references variable.
- *typeRef* – Method references type, including class instance creation expressions and casting conversions. E.g.,
  typeRef ("init Ex bar foo Ex.java",
  "BorderLayout awt java library")
- *genRef* – Method contains undifferentiated references, e.g.,
  genRef ("action Ex bar foo Ex.java",
  "out System lang java library")
- *varType* - Variable has type, e.g.,
  varType ("ev action Ex bar foo Ex.java",
  "Event awt java library")

The facts *funRef, varRef*, *typeRef* and *genRef* are used to denote the as yet unknown entities linking to external entities or the library. They might be inherited members or they might come from qualified references. They will be resolved to their correct UIDs in the next section. So far, from source code we have inferred that each *funRef* refers to an external method, each *varRef* refers to an external field, and so on. But *genRef* facts are ambiguous, meaning that the correct semantics for the reference has yet to be determined – we do not know if the entity referred to is a class type, package, field or method.

Figures 5 and 6 show examples of the data design facts extracted from the library and class files of our sample example program.

3.7 Step 7: Deriving Link Relationships.

Based on the data design model facts recovered from the source and library files in Section 3.6 (Figure 5, Figure 6), we can derive the final linking relationships between object and external entity references and defined entities in the library or other source files of the program. Once we have determined these links, we can replace the temporary approximate UIDs we generated for these references in Section 3.5 with the UIDs of their actual target entities. The link relationships are derived by encoding Java linking rules as inference rules in a relational system such as *Prolog* or *Grok* [24]. *Grok* is a Tarski relational algebra calculator which has been used in manipulating graphs for large scale software systems to assist in program visualization and understanding [10].

The derived links relating the external reference UIDs in our sample program to the UIDs of the actual external class file and library entities are shown in Table 1. In the following subsections we explain how these links are derived using *Grok*.

3.7.1. Linking Through Inherited and Overridden Methods. Some external fields and methods are inherited from superclasses. In the current class, those entities can be used as references which refer to the ones defined in superclasses. For example, *setLayout()* is an inherited method from *java.awt.Container* with the inheritance relationship:

```
java.lang.Object
|
+-- java.awt.Component
   |
   +--java.awt.Container
      |
      +--java.awt.Panel
         |
         +--java.applet.Applet
            |
            +--foo.bar.Ex
```

Input to our *Grok* scripts includes all the data design facts design recovered from all program class files and the library. We begin by identifying all the temporary references that need to be linked. These include both the qualified object references and the qualified external references in all of the renamed class files. The result of this analysis is stored in the relationship *needlink*, which documents all of the temporary UIDs for which we need to resolve a target entity.

We compute the *needlink* relationship using *Grok* to find those tuples in *funRef*, *typeRef, varRef* and *genRef* for which there is no directly defined entity in any program class file or library file. So for our sample example, the inferred *needlinks* are:

needlink "init Ex bar foo Ex.java"
"setLayout Ex bar foo Ex.java"
needlink "action Ex bar foo Ex.java"
"println out System lang java library"
needlink "main Ex bar foo Ex.java"
"resize f main Ex bar foo Ex.java"
needlink "main Ex bar foo Ex.java"
"init ex main Ex bar foo Ex.java"

We now have all the information we need to actually resolve the needed links. First, we use *Grok* to explore the class hierarchy to resolve references to inherited (super class) methods and fields. We encode the results of this analysis as the

```
hasSuperType ("System lang java library", "Object lang java library")
hasMemberForClass ("System lang java library",
            "out System lang java library")
hasFieldType ("out System lang java library",
            "PrintStream io java library")
hasSuperType ("Panel awt java library", "Container awt java library")
hasSuperType ("Applet applet java library", "Panel awt java library")
hasMemberForClass ("Applet applet java library",
            "init Applet applet java library")
hasMethodType ("init Applet applet java library", void)
hasSuperType ("Container awt java library", "Component awt java library")
hasMethodType ("setLayout Container awt java library", void)
hasMemberForClass ("Container awt java library",
            "setLayout Container awt java library")
hasMethodType ("action Component awt java library", boolean)
hasMemberForClass ("Component awt java library",
            "action Component awt java library")
```

Figure 5. A Subset of the Facts for the Java Library.

```
hasSuperType ("Ex bar foo Ex.java", "Applet applet java library")
hasFieldType ("x Ex bar foo Ex.java", int)
hasMethodType ("init Ex bar foo Ex.java", void)
hasMethodType ("action Ex bar foo Ex.java", boolean)
hasMemberForClass ("Ex bar foo Ex.java", "x Ex bar foo Ex.java")
hasMemberForClass ("Ex bar foo Ex.java", "init Ex bar foo Ex.java")
hasMemberForClass ("Ex bar foo Ex.java", "action Ex bar foo Ex.java")
paramVar ("action Ex bar foo Ex.java", "ev action Ex bar foo Ex.java")
localVar ("main Ex bar foo Ex.java", "x main Ex bar foo Ex.java")
hasVarType ("x Ex bar foo Ex.java", int)
funRef ("init Ex bar foo Ex.java", "setLayout Ex bar foo Ex.java")
funRef ("action Ex bar foo Ex.java", "println out System lang java library
funRef ("main Ex bar foo Ex.java", "init ex main Ex bar foo Ex.java")
genRef ("action Ex bar foo Ex.java", "System lang java library")
genRef ("action Ex bar foo Ex.java", "out System lang java library")
typeRef ("main Ex bar foo Ex.java", "Frame awt java library")
varType ("ev action Ex bar foo Ex.java", "Event awt java library")
varType ("arg action Ex bar foo Ex.java", "Object lang java library"
```

Figure 6. A Subset of the Facts for the Sample Program.

superLink relationship. For example, for our sample example program we infer:

superLink “setLayout Ex bar foo Ex.java”
“setLayout Container awt java library”

Next we use *Grok* to explore the class hierarchy to resolve overridden methods. In our sample example, the overridden methods *init()* and *action()* are originally defined in *java.applet.Applet* and *java.awt.Component*. We encode the results of this analysis in the *overriddenMethod relationship*. For example, for our sample program we infer:

overriddenMethod “init Ex bar foo Ex.java”
“init Applet applet java library”
overriddenMethod “action Ex bar foo Ex.java”
“action Component awt java library”

3.7.2. Linking Through Object Instances of Classes. The only remaining unresolved links needed are qualified object references referring to the members of the object’s class or superclasses. To resolve these references, we use *Grok* once again to explore the object’s class hierarchy beginning with the class of the object itself and looking upward until the member referred to is found. We encode the results of this analysis in the *classLink* relationship. In the case of our sample example, we infer the links:

classLink “println out System lang java library”
“println PrintStream io java library”
classLink “init ex main Ex bar foo Ex.java”
“init Ex bar foo Ex.java”
classLink “resize f main Ex bar foo Ex.java”
“resize Component awt java library”

3.8 Step 8: Push Links into Uniquely Renamed Source.

The final step in our unique renaming involves replacing the temporary UIDs used for object and external qualified names in the uniquely renamed source with the UIDs of the corresponding actual external entities inferred by the linking process. This is done using a final TXL source transformation that uses the inferred link facts of the previous section to replace each linked UID instance in the source with the UID that it is really linked to. In the case of our example sample program, we come up with the final uniquely renamed Java source code of Figure 7. The size of the final annotated source code is only about five times larger than the original, even though it contains many times more information.

4. Related Work

The unique renaming paradigm and UID schema on which our work is based was originally designed by Schneider [29] for design recovery and analysis of programs written in the Turing programming language. LS/2000 [9] is a TXL-based process that used the same paradigm in design recovery techniques to analyze source code for Year 2000 risks. It guided source transformations that were able to automatically migrate over 99% of the year 2000 risks in over three billion lines of production IT source written in COBOL, PL/I and RPG. The use of UIDS to link between source code and design databases was further explored through the HSML [31] language. The main contribution of this paper to the LS/2000 work is the extension of the unique naming concept to object oriented language constructs present in Java and not in COBOL, PL/I or RPG. The work in this paper also addresses the more flexible relationship possible between Java source entities that are also not present in the languages supported by LS/2000.

Cox and Clarke [11] developed Jupiter repository system. Maia is a data model that is encoded using XML like markup. Tags are used to mark entities such as blocks, declarations, and control flow. Source tokens are numbered sequentially and the markup tokens are assigned fractional token positions based these source token numbers. Links between tokens are done using attributes that give the source token positions. So a delcaration is annotated up with markup tokens that give the source token positions of the use of the entity while references to tokens are annotated with markup that gives the declaration of the entity. The approach has the flexibility that the tags may be stored in the source code or separately in a design database. The disadvantage of using token numbers in the references is that the markup is more sensitive to changes in the code. Our unique identifier approach will survive transformations to the code.

Middle level models such as the Dagstuhl Middle Model

	Original UID	Linked UID
Inherited members and overridden methods	"*setLayout Ex bar foo Ex.java*" "*init Ex bar foo Ex.java*" "*action Ex bar foo Ex.java*"	"*setLayout Container awt java library*" "*init Applet java library*" "*action Component awt java library*"
Member invocations from class instances	"*println out System lang java library*" "*init ex main Ex bar foo Ex.java*" "*resize f main Ex bar foo Ex.java*"	"*println PrintStream io java library*" "*init Ex bar foo Ex.java*" "*resize Component awt java library*"

Table 1. Derived Links for External Reference UIDs

(DMM) [30] encode the source position of entities in the model. In DMM, model objects are associated with source objects via the defines and/or the declares relation. The source objects in the design database have an identity of thier own and are linked to the source code by attributes defining the start and end position as line and column numbers. Datrix [32,33], an abstract syntax graph approach, also stores the source code locations directly in the model as line and column attributes. Both of the DMM and Datrix approaches share the disadvantage of Maia. One could argue that the ASG based models do not need the link to the source code other than for reporting purposes, since they are source code complete. Any transformations could be done entirely in the design database. However, the the fixed schema (i.e. fixed grammar) of the design database limits some techniques that can be used to simplify transformations [34].

There are many papers that explore how to represent source code information for different languages in XML format. Power and Malloy[2] modify the GNU bison parser generator to generate parse trees in XML format for C, Objective C, C++, Java and FORTRAN. Another program analysis tool, XMLizer [3], also outputs XML format to represent program structure for Java, PL/IX and Pascal. In both cases the XML is used to represent the parse tree, and does not contain any attributes linking the use of an identifier to its delcaration. Power and Malloy absorb all of the source text into tags and attributes. XMLizer has the ability to represent partial parse trees. For example, a statement non-terminal may mark text for an entire statement with no parse reprentation embedded in the statement. When fully parsed, most text is absorbed into XML tags and attributes, although constant and identifiers remain as marked up

```
package <UID id="bar foo Ex.java ">foo.bar </UID> ;
import java.applet .*;
import java.awt .*;
public class <UID id="Ex bar foo Ex.java ">Ex </UID> extends <UID id="Applet applet java library"> Applet </UID> {
  public static int <UID id="x Ex bar foo Ex.java ">x</UID> = 100;
  public static void <UID id="x Ex bar foo Ex.java ">x</UID> () {
    int <UID id="y x Ex bar foo Ex.java ">y </UID> ;
    <UID id="y x Ex bar foo Ex.java ">y </UID> = <UID id="x Ex bar foo Ex.java ">x </UID> ;
    <UID id="x Ex bar foo Ex.java ">x</UID> = <UID id="x Ex bar foo Ex.java ">x</UID> * 10;
  }
 public void <UID id="init Applet applet java library"> init </UID> () {
    <UID id="setLayout Container awt java library"> setLayout ( new <UID id="BorderLayout awt java library"> BorderLayout </UID> () ) </UID> ;
 }
 public boolean <UID id="action Component awt java library"> action </UID> (
     <UID id="Event awt java library"> Event </UID> <UID id="ev action Ex bar foo Ex.java ">ev </UID> ,
       <UID id="Object lang java library"> Object </UID> <UID id= "arg action Ex bar foo Ex.java ">arg </UID> ) {
     <UID id="println PrintStream io java library"> <UID id= "out System lang java library"> <UID id="System lang java library"> System </UID>
      . Out </UID> . println ( <UID id="x Ex bar foo Ex.java ">x</UID> ) </UID> ;
     return true;
 }
public static void <UID id= "main Ex bar foo Ex.java ">main </UID> ( <UID id="String lang java library"> String </UID>
     <UID id="args main Ex bar foo Ex.java ">args </UID> [] ) {
     int <UID id="x main Ex bar foo Ex.java ">x</UID> = 120;
     int <UID id="y main Ex bar foo Ex.java ">y</UID> = 100;
     <UID id="x Ex bar foo Ex.java ">x () </UID> ;
     if ( <UID id="y main Ex bar foo Ex.java ">y</UID> < <UID id="x main Ex bar foo Ex.java ">x</UID> )
           <UID id="y main Ex bar foo Ex.java ">y </UID> = <UID id="x main Ex bar foo Ex.java ">x</UID> ;
     <UID id="Frame awt java library"> Frame </UID> <UID id="f main Ex bar foo Ex.java ">f </UID> =
        new <UID id="Frame awt java library"> Frame </UID> ("C");
    <UID id="resize Component awt java library"> <UID id= "f main Ex bar foo Ex.java ">f </UID> . resize(210, 200 )</UID> ;
    <UID id="Ex bar foo Ex.java ">Ex </UID> <UID id= "ex main Ex bar foo Ex.java ">ex</UID> = new <UID id="Ex bar foo Ex.java ">Ex </UID> ();
    <UID id="init Applet applet java library"> <UID id= "ex main Ex bar foo Ex.java ">ex</UID> . init() </UID> ;
  }
}
```

Figure 7. Final Linked Renamed Java Source Code.

text. JavaML [14] uses a similar representation as Power and Malloy, but includes a unique identifier in each attribute. This attribute links variables and methods within a file. Methods between files are not attributed.

There are many other tools for source analysis of Java. Sun's JavaCheck [15] can analyze the use of library APIs for compatibility. SHriMP [16, 17, 18], Chava [19], GUPRO [20, 21] and the Software Bookshelf [22] are tools that can extract and visualize information from Java programs.

5. Conclusions and Future Work

We have described a unique renaming system for Java programs that accurately resolves relationships between program entities in source using unique names (UIDs). Each declared name and reference is annotated with its unique name in the source using XML markup. The UIDs serve as keys uniquely identifying each program entity in both the source and the design database. They form a kind of bridge between the two which allows for independent processing of both source and design without losing the connections between the two.

Our unique renaming is implemented using a sequence of source transformations written in the TXL language. Data design recovery from the initial renamed source yields a set of base facts used as input to a *Grok* script to infer links between external references and the appropriate external entities. Derived links are reflected back into the renamed source using a final source transformation, resulting in a fully linked uniquely named source representation of the program suitable for complex program comprehension, analysis, visualization and transformation tasks.

Renamed Java programs contain far more information than their original source. Both declarations and references are clearly marked with the globally unique UID of the entities they refer to, freeing further analysis from worrying about ambiguities. Renamed code can be easily parsed at different levels (light, middle or heavy weight) as either Java source or an XML document. Because unique naming is represented entirely as XML markup of original source text, output of subsequent analysis or transformation tasks can easily include or exclude UIDs in their results. Unique renaming can be easily integrated to other reverse engineering tools. For example, very little modification of our data design facts would allow them as input to Rigi [28] or Moonen's code smell detection process [12].

The unique renaming described in this paper has thus far been used for only one actual application, a system to assist in Java library version migrations. Using unique renaming of different versions of the AWT library and source programs using it, an accurate analysis of AWT version dependencies and migration path was easily derived. The technique is completely generic and can be used for any other library version migration.

In future work, we may consider adding resolution to distinguish overloaded methods with different UIDs. Also, thus far our method is based entirely on static analysis. For some applications it would be more useful if information from dynamic analysis were added as well. For example, Java reverse engineering projects based on Java bytecode [25, 26] could provide more facts to enrich the data model.

We believe that unique renaming is a very basic and important step in Java design recovery and analysis. For example, renamed Java code already has the class dependency information necessary to derive UML or other representations of the program design. We hope to explore and exploit the properties of uniquely renamed code to make more effective use of existing analysis tools and techniques in the coming years.

References.

[1] World Wide Web Consortium. Extensible Markup Language (XML). *http://www.w3.org/xml/*

[2] J. F. Power and B. A. Malloy. Program annotation in XML: a parse-tree based approach. *9th Working Conference on Reverse Engineering (WCRE 02),* pp. 190-198, Oct. 2002.

[3] G. McArthur, J. Mylopoulos, and S. K. K. Ng. An Extensible Tool for Source Code Representation Using XML. *9th Working Conference on Reverse Engineering (WCRE 02),* pp.199-208, Oct. 2002.

[4] A. Asencio, S. Cardman, D. Harris, and E. Laderman. Relating Expectations to Automatically Recovered Design Patterns. *9th Working Conference on Reverse Engineering (WCRE 02),* pp. 87-96, Oct. 2002.

[5] Claudio Riva and Yaojin Yang. Generation of Architectural Documentation using XML. *9th Working Conference on Reverse Engineering (WCRE 02),* pp. 161-169, Oct. 2002.

[6] E. Mamas, K. Kontogiannis. Towards Portable Source Code Representations Using XML. *7th Working Conference on Reverse Engineering (WCRE 00),* pp. 172-182, Nov. 2000.

[7] TXL Project, The TXL Programming Language, Version 10.2. *http://www.txl.ca/docs/TXL102LangRef.pdf*, Apr. 2002.

[8] J.R. Cordy, T.R. Dean, A.J. Malton and K.A. Schneider. Source Transformation in Software Engineering using the TXL Transformation System. Special Issue on Source Code Analysis and Manipulation, *Journal of Information and Software Technology,* Oct. 2002.

[9] T.R. Dean, J.R. Cordy, K.A. Schneider and A.J. Malton. Experience Using Design Recovery Techniques to Transform Legacy Systems, *IEEE International Conference on Software Maintenance(ICSM 2001),* pp. 622-631, Nov. 2001.

[10] R.C. Holt. Structural Manipulations of Software Architecture Using Tarski Relational Algebra. *5th Working Conference on Reverse Engineering (WCRE 98),* pp. 210-219, Oct. 1998.

[11] A. Cox and C. Clarke. Representing and Accessing Extracted Information. *IEEE International Conference on Software Maintenance(ICSM 2001),* pp. 12-21, Nov. 2001

[12] E. van Emden and L. Moonen. Java Quality Assurance by Detecting Code Smells. *9th Working Conference on Reverse Engineering (WCRE 02),* pp. 97-106, Oct. 2002.

[13] J. Gosling, B. Joy, G. Steele and G. Bracha. Sun Microsystems Inc. The Java Language Specification (2nd edition). Addison Wesley, 2000.

[14] G.J. Badros. JavaML: A Markup Language for Java Source Code. *9th International World Wide Web Conference,* pp. 159-

177, May 2002.

[15] Sun Microsystems Inc. JavaCheck - Platform Compatibility Insurance for your Applications and Applets. *http://java.sun.com/products/personaljava/javacheck.html*.

[16] M.-A. D. Storey, H. A. Müller and K. Wong. Manipulating and Documenting Software Structures. Series on Software Engineering and Knowledge Engineering, Vol. 7 Software Visualization, pp. 244-263, Nov. 1996.

[17] J. Michaud, M.-A. Storey and H. Muller. Integrating Information Sources for Visualizing Java Programs. *IEEE International Conference on Software Maintenance(ICSM 2001),* pp. 250-259, Nov. 2001.

[18] University of Victoria. SHriMP Views. *http://shrimp.cs.uvic.ca/*.

[19] J. Korn. Chava: Reverse Engineering and Tracking of Java Applets. *6th Working Conference on Reverse Engineering (WCRE 99),* pp. 314-325, Oct. 1999.

[20] C. Lange, H. M. Sneed and A. Winter. Comparing graph-based program comprehension tools to relational database-based tools. *9th International Workshop on Program Comprehension (IWPC 01),* pages 209-218, May 2001.

[21] GUPRO - Generic Understanding of PROgrams. *http://www.uni-koblenz.de/~ist/gupro.en.html*.

[22] J.!Finnigan, R.C.!Holt, I.!Kalas, S.!Kerr, K.!Kontogiannis, H.A. Müller, J.!Mylopoulos, S.G.!Perelgut, M.!Stanley, K.!Wong. The Software Bookshelf. *IBM Systems Journal* 36(4), pp.564-593, Nov.1997

[23] TXL Project, The TXL Grammar Collection. *http://www.txl.ca/nresources.html*.

[24] R.C. Holt. Introduction to the Grok Language. *http://plg.uwaterloo.ca/~holt/papers/grok-intro.html*.

[25] L. A. Barowski and J. H. Cross II. Extraction and Use of Class Dependency Information for Java. *9th Working Conference on Reverse Engineering (WCRE 02),* pp. 309-315, Oct. 2002.

[26] D. Rayside, S. Kerr and K. Kontogiannis. Change and adaptive maintenance detection in Java software systems. *5th Working Conference on Reverse Engineering (WCRE 98),* pp. 10-19, Oct. 1998.

[27] Sun Microsystems Inc. The Java Language: An Overview. *http://java.sun.com/docs/overviews/java/java-overview-1.html*.

[28] M.-A. D. Storey, K. Wong, and H. A. Mller. Rigi: A visualization environment for reverse engineering. *19th International Conference on Software Engineering (ICSE '97),* pp. 606--607, May 1997.

[29] J.R. Cordy and K.A. Schneider, Architectural Design Recovery Using Source Transformation, *1995 Conference on Computer Aided Software Engineering (CASE `95) Workshop on Software Architecture,* Toronto, July 1995.

[30] T.C. Lethbridge *et al.*, The Dagstuhl Middle Model version 0.005,!*http://scgwiki.iam.unibe.ch:8080/Exchange/uploads/2/DMMDescriptionV0005.pdf*

[31] J.R. Cordy, K.A. Schneider, T.R. Dean and A.J. Malton. HSML: Design Directed Source Code Hot Spots. *9th International Workshop on Program Comprehension (IWPC 2001)*, pp. 145-154, May 2001.

[32] Bell Canada. DATRIX™ Abstract Semantic Graph Refernece Manual Version 1.4. Bell Canada, Inc., Montreal, May 2000.

[33] T.R. Dean, A.J. Malton and R.C. Holt. Union Schemas as a Basis for a C++ Extractor. *8th Working Conference on Reverse Engineering (WCRE 2001)*, pp. 59-67, Oct. 2001.

[34] T.R. Dean, J.R. Cordy, A.J. Malton and K.A. Schneider. Grammar Programming in TXL. *IEEE 2nd International Workshop on Source Code Analysis and Manipulation (SCAM 2002),* pp. 93-102, Oct. 2002.