2014 IEEE 15th International Symposium on High-Assurance Systems Engineering

Using Program Transformation, Annotation, and Reflection to Certify a Java Type Resolution Function

Victor L. Winter, Carl Reinke, Jonathan Guerrero
Department of Computer Science
University of Nebraska at Omaha
Omaha, Nebraska
USA
{vwinter, creinke, jguerrero}@unomaha.edu

***Abstract*—In Java, *type resolution* is a function that takes a reference to a type occurring in a given context as input, and returns the canonical form of that type. This information is fundamental to static analysis – a "must have" function underlying virtually all forms of semantic-based analysis. In the case of Java, this function is also complex and it is quite common to encounter tools where it is implemented incorrectly.**

This paper presents a novel approach for certifying the correctness of a given type resolution function with respect to an arbitrary Java source code base. The approach uses *program transformation* to instrument a subject code base in such a way that *reflection* can then be used to certify the correctness of the type resolution function against the function used by the Java compiler. In this form of certification, the type resolution function of the Java compiler serves as the test oracle.

***Keywords*-source code analysis; Java; type resolution; reflection; annotation; program transformation;**

I. Introduction

In Java, *type resolution* – the determination of the canonical form of a type reference with respect to a given context – requires complex analysis. A simple example highlighting the essence of the type resolution problem is shown in Figure 1.

```
package p1;

public class A {
    class B1 {}
    class innerA extends B {
        B1 myB1; // What is the canonical name
                 // of the type reference B1?
                 // Is it p1.A.B1 or p1.A.B.B1?
    }


    class B {
        private class B1 {}
    }
}
```

Figure 1. Type Resolution

In practice, the vast majority of type resolution instances can be trivially resolved. However, a number of corner cases do exist in which resolution is non-trivial. We have developed a type resolution test suite whose complexity supports this claim. Specifically, our test suite attempts to exercise the full range of resolution possibilities within Java. This test suite has revealed bugs in the static analysis performed by the Eclipse, Netbeans, and IntelliJ editors as well as their refactoring tools [10]. However, the story does not end there: Figure 2 shows a small Java program that when compiled from the command line using javac 1.6.0_26, will result in a stack overflow.[1] We present these items as evidence that, for the Java language, the resolution of types (and identifiers) is indeed non-trivial.

```
package stackoverflow;
import static stackoverflow.A00.*;
public class A00 extends B00
{
    public static class B00{}
}
```

Figure 2. A small program causing a stack overflow in the Java 1.6.0_26 compiler.

Irrespective of its complexity, type resolution is fundamental to static analysis – a "must have" function for virtually all forms of semantic-based analysis. When developing a source code analysis and manipulation tool, there are two basic approaches to obtaining resolution-based information: (1) extract resolution information from the Java compiler (e.g., from class files), or (2) implement a resolution function.

For systems that implement a resolution function, certification of the function's correctness is *critical* – especially in light of the evidence presented in this section.

A. Contribution

This paper describes a novel approach for certifying a given Java type resolution function. The approach presented uses a set of *program transformations* to instrument Java source code by adding additional field declarations as well

[1]This bug has been fixed in Java 1.7.0_03.

978-1-4799-3466-9/14 $31.00 © 2014 IEEE
DOI 10.1109/HASE.2014.27

as *annotations* containing results obtained from the resolution function under test. As part of the transformation process, a certification method `__validate()` is also created containing reflective hooks to all types declared within the computation unit. This method is then added, via transformation, to the primary type of the compilation unit. This process is repeated for each compilation unit (aka file) within the Java source code base driving the certification. And finally, a test engine is created that is tuned to the test at hand. The result is a certification system in which *reflection* can be used to check the correctness of the targeted resolution function's behavior with respect to the instrumented code base.

A sample of the output produced from a certification is shown in Figure 3. In this manner, the resulting system can be used to certify, in a fully automated fashion, that the analysis produced by the targeted type resolution function is in agreement with the analysis produced by the Java compiler. The assumption underlying this form of certification is that the semantics of resolution is implicitly defined to be "*what the compiler says it is*". That is, the type resolution function of the Java compiler serves as an oracle.

```
Test result: PASSED

Files inspected = 70
Classes inspected = 350

References correctly resolved = 380
References incorrectly resolved = 0
Unresolved references = 0


Canonical names processed:
     p001.A
     p001.B
     p001.Bucketx29
     p002.A
     ...
```

Figure 3. Sample certification output.

The remainder of the paper is laid out as follows: Section II takes a more detailed look at the type resolution problem and presents examples supporting the claim that this problem is hard. Section III describes our approach. Section IV discusses results as well as current limitations of our system. Section V overviews related work. And Section VI concludes.

II. Type Resolution

This section gives an overview of the type resolution problem, highlighting some of its complexities. Due to space constraints, our discussion of type resolution here is an approximation. A number of low-level technical details are omitted. However, the details that we do provide are hopefully sufficient to give the reader a feel for the complexity of the problem. More details on this subject can be found in [5][4].

In Java, a *reference* to a type or field consists of one or more simple identifiers separated by dots.

$$id_1.id_2.\cdots.id_n$$

The goal of *resolution* (type resolution as well as field and method resolution) is to convert such a reference into a *canonical form* – a term which has a similar structure, but which describes the location of the item being referenced in absolute, fully qualified, terms (e.g., `java.lang.Object`).

Resolution proceeds in an incremental fashion by resolving the simple identifiers belonging to a reference in a left-to-right fashion. Each incremental step produces a result (e.g., a type), called a *resolvent*, which is then used as the starting point for the incremental step that follows. In such a resolution sequence, the resolution of the primary identifier(s) is distinct from all the others. In approximate terms, the following steps are followed when resolving the simple identifiers within a reference.

1) **Resolution of the primary identifier(s)** id'**.** Starting from the type (e.g., class) in which the reference is located do the following:
   a) Search the current type for a matching field (or method) declaration.
   b) If not found: Search up the hierarchy for a matching field (or method) declaration.
   c) If not found: Search static imports of the compilation unit for fields and types with a preference given to fields in contexts where both a field and a type could occur.
   d) If not found: Search hierarchy for a matching type declaration.
   e) If not found: Search the imports of the compilation unit for a matching *single-type* import.
   f) If not found: Search all compilation units of the current package for a matching type declaration.
   g) If not found: Search the imports of the compilation unit for a matching *on-demand* import. The contents of the package `java.lang` is implicitly included as an on-demand import.
   h) If not found: Search the project for a matching package. If this occurs, continue to process the simple identifiers of the reference until a type identifier is encountered (e.g., `java.lang.Object`).
   i) If not found: Fail.
2) **Resolution of the secondary identifiers** id''**.** Starting from the primary resolvant, r_1, obtained from the

previous step, to resolve id'' do the following:

a) If the resolvant r_1 corresponds to a field, then do the following:
   i) Obtain the canonical form for the type of the field and search the hierarchy of this type for a field matching id''.

b) If the resolvant r_1 corresponds to a type, then do the following:
   i) Search the inheritance hierarchy of the type for a member declaration (which could be a field or a type) matching id''.

A. Static Imports

Static imports were introduced into Java in version 5.0, and provide a mechanism for importing specifically designated members of a class that are `static`. The original intention was to use this as a mechanism for importing constants (e.g., π) without having to import the entire class in which these constants were defined. However, since nested types also constitute class members, this opened the door to potential ambiguities concerning exactly which members of a class are being imported.

Figure 4 shows an example of the complexity that can be encountered when combining the static import mechanism with the resolution rules given in Section II. This example was developed for this paper, but coincidentally (and quite unintentionally) also exposed another bug in Eclipse.

```
// ================================================
package p1;

import static p2.B.X; // imports both a field
                      // and a type.
public class A {
    int x = X;
    X myX; // Eclipse Version: Helios
           // Release Build id: 20100617-1415
           // fails to resolve: error
}

class A1 extends X {}

// ================================================
package p2;

public class B {
    public static int X;
    public static class X {}
}
```

Figure 4. An example revealing a bug in Eclipse.

B. Nested Types

Nested types are type definitions that occur within other type definitions, methods, constructors, or expressions involving object instantiations. There are four nesting classifications for a type T_1.

- **static member types** – If T_1 is an interface, enum, or annotation type, then T_1 is implicitly a static type. If T_1 is a class, then it must be explicitly designated as `static` in order for it to be classified as a static type. If T_1 is a static type and resides within another type T_2, then T_1 is a *static member type* of T_2.
- **non-static member classes** – If T_1 is a class that is not declared static and T_2 is a class or enum, then T_1 is a *non-static member class* of T_2.
- **local classes** – If T_1 is a non-static class occurring within method or constructor, then T_1 is a *local class*.
- **anonymous classes** – If T_1 is a nameless class (i.e., a class body) occurring within the context of an object instantiation, then T_1 is an *anonymous class*.

Historically speaking, nested types were not part of the original design of Java. They were added in Java 1.1 and are responsible for a considerable amount of complexity in the resolution algorithm. For example, nested classes are part of class hierarchies which can be distinctly different (or overlapping) with the subtype hierarchies of the outer classes in which they reside. This gives rise to subtle opportunities for classes to inherit protected members from other classes within which they are nested.

Nested classes also obfuscate some of the intuitions regarding visibility. For example, even though a nested class has visibility over the private members of a sibling nested class, it cannot inherit private members from that sibling. This is the root cause of some errors in Eclipse and IntelliJ [10].

Aside: From an operational standpoint, it is worth mentioning that the introduction of nested types did not change the JVM. During compilation, all types are flattened leaving only top-level types [4]. The Java compiler accomplishes this by inserting hidden fields, methods, and constructor arguments (as needed) into the newly generated types. The *javap* disassembler can be used to get a clearer picture of what the compiler actually does in such cases.

C. Access Control

Visibility of types and type members can be understood using standard abstractions drawn from the security arena. Specifically, a protection state can be associated with a reference. This protection state stores the visibility attributes over which a reference has access privileges. The protection state can be modeled as a tuple (p_1, p_2) consisting of a *non-inheritance based* protection state p_1, and an *inheritance based* protection state p_2. The initial state of p_1 associated with a reference grants access to members having `public`, `protected`, *package-private*, and `private` visibility attributes. The inheritance based portion of the protection

state p_2 concerns itself exclusively with additional visibility associated with the `protected` attribute.

Structural relationships, as defined by the location of resolvants within a Java project, must be tracked during resolution. Structural properties of interest with respect to (non-inheritance based) access control can be stated as follows. Given two adjacent resolvent types, r_i and r_{i+1}, determine which of the following property holds:

- **intraclass**(r_i, r_{i+1}) – r_i and r_{i+1} are located within the same (top-level) class.
- **intrapackage**(r_i, r_{i+1}) – r_i and r_{i+1} are located within the same package and $\neg$intraclass(r_i, r_{i+1}).
- **interpackage**(r_i, r_{i+1}) – r_i and r_{i+1} are located in distinct packages within the project.

Rules based on these properties govern the transitions of p_1, which are monotonic (e.g., once p_1 gives up access to the `private` attribute, it cannot get it back).

Transitions of the inheritance based portion of the protection state p_2 is governed by rules that run orthogonal to those governing p_1. For example, access over the `protected` attribute is not subject to removal when resolvants transition across class or package boundaries. Conceptually speaking, a protected member of a type T_1 is visible from within a type T_2 if $T_2 <: T_1$. This idea becomes more complex in the presence of nested types. The reason being that the protected visibility of an enclosing type is also passed on to its nested types. Additional complexity is added as a result of the fact that an identifier within a reference can denote either a field or a type.

Figure 5 shows some of the difficulties encountered when it comes to determining visibility over `protected` members of a type. In particular, note that in the nested class `p1.A.nestedA`, the references `e.Bi` and `c.e.Bi` are not resolvable. More specifically, in both cases it is the identifier `Bi` that is not visible even though the class `p3.E` extends the class `p2.B`.

```
// ================================================
package p1;

import p3.C;
import p3.D;
import p3.E;

public class A extends p2.B {
    public class nestedA {
        C c;
        D d;
        E e;

        int x1 = c.Bi;
        int x2 = d.c.Bi;
        int x3 = e.Bi;  // not visible
        int x4 = c.e.Bi; // not visible
    }
}

// ================================================
package p2;

public class B {
    protected int Bi;
}

// ================================================
package p3;

public class C extends p1.A {
    public E e;
}

// ================================================
package p3;

public class D {
    public C c;
}

// ================================================
package p3;

public class E extends p2.B {}
```

Figure 5. An example of the complexity of inheritance-based access control.

III. Automated Certification

We hope that the discussion in the previous section has convinced the reader that resolution in Java is indeed non-trivial, and that the tests that exercise idiosyncrasies of resolution can be esoteric. In practice, a wide spectrum of tests need to be developed in order to cover Java's resolution behavior in a comprehensive fashion. The consequent challenge is how to then certify that an "in-house" developed resolution algorithm has the same behavior as that of the Java compiler with respect to the given test suite (or some other code base for that matter).

The domain of discourse we assume (i.e., in which certification of type resolution needs to occur) is one in which static analysis tools are undergoing continuous evolutionary change. Tool enhancements are being made, underlying functionality is being re-implemented, and code and data structures are being refactored and refined. Under these conditions it is extremely beneficial to develop automated testing capabilities for the (re)certification of system functionality – especially critical functionality such as resolution.

In our approach, to certify that a given type resolution function conforms to that of the Java compiler one must do the following:

1) Assemble a set S of Java programs P_i exercising type resolution:

$$S = \{P_1, P_2, \ldots, P_n\}$$

At the system level, each P_i is a folder hierarchy containing compilation units $CU_{i,j}$ (i.e., dot-java files).

$$P_i = \{CU_{i,1}, CU_{i,2}, \ldots, CU_{i,n_i}\}$$

In theory, S can be any collection of Java source programs (e.g., a particular application for which one wants to confirm that the type resolution function behaves correctly). However, to provide a more comprehensive certification, the construction of S should be more systematic with respect to type resolution possibilities.

2) Instrument the source code in S producing:

$$S' = \{P'_1, P'_2, \ldots, P'_n\}$$

3) Generate a test execution engine $\texttt{Certify}_{S'}$ corresponding to S'.
4) Compile and execute $\texttt{Certify}_{S'}$.

It is beyond the scope of this paper to discuss techniques for the creation of S. Instead, we focus on steps 2 and 3.

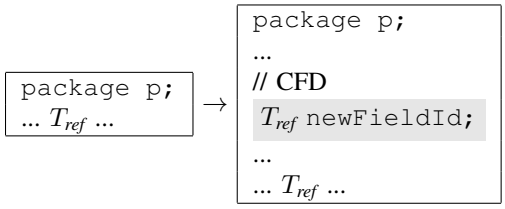
A. Source Code Instrumentation

Source code instrumentation assumes that

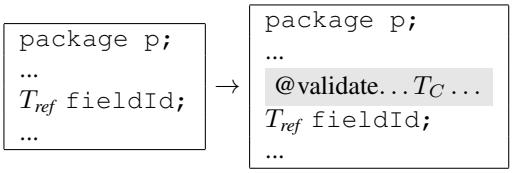
$$S = \{P_1, P_2, \ldots, P_n\}$$

is given, and has three basic goals:

1) Create for every (non-generic) type reference T_{ref}, occurring within a compilation unit $CU_{i,j}$, a *corresponding field declaration (CFD)* whose type resolution is equivalent to the original type reference T_{ref}.



2) Add an annotation containing T_C to *every* (non-generic) field declaration (CFD or otherwise), where T_C is the canonical form of T_{ref} as determined by the type resolution function under test.



3) Add a method `__validate()` to the primary (i.e., `public`) class of each compilation unit. The body of this method consists of a call to a reflective validator. The arguments passed to this validator are the Class Objects (e.g., `A.class`) for every type declared within the compilation unit.

Figure 6 shows a field declaration annotated with the canonical type name `example01.A.C.C1`.

```
@validation.Validator(
    author = "automatic",
    date = "2012-04-11",
    type = "example01.A.C.C1"
)
C.C1  myC3  ;
```

Figure 6. An example of a field annotated with the canonical type name `example01.A.C.C1`.

There are two primary challenges to the transformation-based instrumentation described: (1) invoking the type resolution function with the proper context – this requires canonical names of the contexts in which type references occur to be tracked during transformation (a topic beyond the scope of this paper which is described in [15]), and (2) positioning CFDs in such a manner that their type resolution will be equivalent to their corresponding type references.

Type references can occur in the following settings:

- **Within types:**
  - The declared type of a field.
  - The super type (i.e., `extends`) of a class.
  - The interface types (i.e., `implements`) of a class.
  - The super types (i.e., `extends`) of an interface.
- **Within methods or constructors:**
  - The types of the formal parameters to a method or constructor.
  - The return types of a method.
  - The types of local variables. This includes both static and non-static initialization blocks.
- **Within expressions:**
  - The types occurring within expressions that perform explicit casting operations.
  - Types implicitly referenced via a constructor call (e.g., `new A()`).

A comprehensive certification of type resolution must account for the occurrence of type references in all of the previously mentioned settings. The generation and placement of CFDs within a code base is governed by the following rules:

1) A generic type does not generate a CFD. The remaining rules pertain exclusively to non-generic types.
2) A field declaration can be directly annotated and does not require the generation of a CFD.
3) If a type reference T_{ref} occurs in an expression initializing a field f, then generate a CFD and add it to the member list in which the declaration of f is contained.
4) If a type reference T_{ref} occurs within a method/constructor, then generate a CFD and add it to the member list of the type (e.g., class) containing the method-/constructor. (Note that the equivalence of the type reference T_{ref} in the CFD with that of the original type reference assumes that methods/constructors do not

contain local class declarations. This is a precondition that our current approach assumes and enforces[2].)

5) If a type reference T_{ref} `extends` or `implements` a static/non-static member type declaration (i.e., a nested type), then generate a CFD and add it to the member list containing the member type declaration.
6) If a type reference T_{ref} `extends` or `implements` a top-level type declaration, then
   a) generate a fresh top-level class declaration C_{fresh} (this only needs to be done once per compilation unit),
   b) add C_{fresh} to the compilation unit in which T_{ref} resides,
   c) generate CFD for T_{ref}, and
   d) add the generated CFD to the member list of C_{fresh}.

B. Test Execution Engine

An instrumented code base

$$S' = \{P'_1, P'_2, \ldots, P'_n\}$$

with

$$P'_i = \{CU'_{i,1}, CU'_{i,2}, \ldots, CU'_{i,n_i}\}$$

can be tested through the execution of a Java program called $\texttt{Certify}_{S'}$. `Certify` is a problem-independent type resolution test program that we have developed to conduct certification tests. During the source code instrumentation phase, problem-specific information is embedded in `Certify` yielding $\texttt{Certify}_{S'}$. More specifically, problem-specific information within $\texttt{Certify}_{S'}$ is an array of triples containing:

1) The classpath associated with a specific test case P'_i.
2) The name of a compilation unit $CU_{i,j}$ residing in P_i.
3) The canonical name of the type, occurring within $CU_{i,j}$, in which the `__validate()` method resides.

C. Enabling Technology

To instrument source code, we use a tool called *Sextant*. *Sextant* is a Java source-code analysis tool under development at the University of Nebraska at Omaha (UNO). From an implementation standpoint, Sextant represents a non-trivial extension of the *TL system* [16][13] (a general-purpose program transformation system) specialized to the domain of the Java programming language.

IV. Results

The system described in this paper is operational. Figures 7 and 8 show a Java source code program before and after instrumentation. Notice that the prefix of field a identifier generated during the creation of a CFD indicates the origin of the field's type reference (e.g., `extensionOfClass_A...`, `fromMethod_f_param...`, and so on). Also notice that instrumentation adds a new class `Bucketx1` to the compilation unit. The field in this class has a type reference corresponding to the `extends` type of `class A`.

```
package example;

public class A extends B {
        C1 f (C2 arg) {
                C1 myC1 = new C1();
                return myC1;
        }

        class nestedA extends innerB {
                C1 g (C2 arg) {
                        C1 myC1 = new C1();
                        return myC1;
                }
        }

}

class B {
        class innerB {}
        class C1 {}
}

class C1 {}
class C2 {}
```

Figure 7. Sample Java source code to be instrumented.

A. Limitations

Currently, our system does not handle type references occurring within anonymous classes or within local classes. Extending our approach to handle these cases requires (1) the creation of new classes corresponding to anonymous classes, and (2) moving local classes out of methods and constructors. The creation of new classes corresponding to anonymous classes is relatively straightforward and we have implemented such a transformation. The movement of local classes out of methods and constructors is more complex. However, guidance for this can be obtained from the Java compiler[4].

V. Related Work

The use of transformation/rewritting to support analysis of a wide range of software artifacts has been explored in a variety of contexts. The use of reflection and annotation to facilitate testing has also been extensively developed. However, we are not aware of any research focusing on the combination of these concepts.

A. Use of Reflection in Testing

Reflection in the context of OO testing is widely used. Frameworks such as Spring, JMockit and JUnit all have unit testing components that make use of reflection. For

[2]In practice, local class declarations are extremely rare.

```
package example;
class Bucketx1
{
    @validation.Validator(author = "automatic", date = "2012-04-13", type = "example.B")
     B  extensionOfClass_A_1  ;
}
public class A extends B
{
    public static boolean __validate()
    {
      return validation.ClassValidator.validateClasses(Bucketx1.class,A.class,B.class,C1.class,C2.class);
    }

    C1 f( C2 arg )
    {
          C1 myC1 = new  C1 ()  ;
         return myC1;
    }

    @validation.Validator(author = "automatic", date = "2012-04-13", type = "example.B.C1")
     C1  fromMethod_f_Body_4  ;

    @validation.Validator(author = "automatic", date = "2012-04-13", type = "example.C2")
     C2  fromMethod_f_Param_3  ;

    @validation.Validator(author = "automatic", date = "2012-04-13", type = "example.B.C1")
     C1  fromMethod_f_Return_2  ;

    class nestedA extends innerB
    {
        C1 g( C2 arg )
        {
              C1 myC1 = new  C1 ()  ;
             return myC1;
        }

        @validation.Validator(author = "automatic", date = "2012-04-13", type = "example.B.C1")
         C1  fromMethod_g_Body_7  ;

        @validation.Validator(author = "automatic", date = "2012-04-13", type = "example.C2")
         C2  fromMethod_g_Param_6  ;

        @validation.Validator(author = "automatic", date = "2012-04-13", type = "example.B.C1")
         C1  fromMethod_g_Return_5  ;
    }
    @validation.Validator(author = "automatic", date = "2012-04-13", type = "example.B.innerB")
     innerB  superTypeOfClass_nestedA_8  ;
}
...
```

Figure 8. Sample Java source code after instrumentation.

example, in JUnit reflection is used to obtain the `Class` object of the type to be tested and then passes this object to the (unit) tester (e.g., `runClasses`). Reflection is also used to find *test methods* (e.g., testXxx()) within a test case[3]. In [11] a tester library is created in which reflection is used to realize structural equality comparisons on objects. In this framework, annotations are used to (1) tag classes containing test methods, and (2) tag test methods thereby relaxing the requirement that test method identifiers begin with the prefix "test".

[3]In JUnit, a test case is a class derived from `TestCase` which contains *test methods*.

Rosa and Martins have developed a fault-injection tool called FIRE (Fault Injection using a Reflective Architecture) [8] in which reflection is used to both inject faults into a C++ software system under test and monitor the systems reaction to the injected faults. This was accomplished using *message interception* mechanism provided by the OpenC++1.2 metaobject protocol. This mechanism enables interception of (1) the actual parameters of a reflective method call, (2) the return value of a reflective method, and (3) access or assignment to a reflective attribute (i.e.,field).

McCaffrey presents a lightweight approach for reflection-based UI testing [9]. The approach supports automated UI tests where reflection is used to (1) launch the application

under test, (2) simulate user moving and resizing actions, (3) validate the application state to determine whether a simulated move or resizing action passed or failed, (4) simulate control actions such as a user typing entering text, (5) validate control actions, and (5) invoke application methods.

B. Use of Transformation in Analysis

Transformation and rewriting has been used to *capture*, *create*, and *check* behavior.

Capture: DMS [3] is a commercial software engineering toolkit supporting program transformation as well as other forms of analysis and manipulation. In [2] an approach is presented demonstrating how transformation can be used to instrument C source code with the goal of capturing branch coverage data associated with a particular test set. (Similarly, in the approach presented in this paper, we use transformation to capture the behavior of a type resolution function, which is then checked via reflection.) In another DMS-based effort [12], source-to-source transformation is used to create subtypes of C++ templates and replace declarations involving the original type with the newly created subtype. The purpose of creating the new subtype is to provide a mechanism by which standard aspected-oriented programming techniques can then be used to isolate instantiations of templates (e.g., assigning a logging behavior only to integer instances of a template). Within the created subtype, overriding methods are declared to which advice can be woven. Thus, this form of transformation can be seen as enhancing an AOP framework, providing support for improving the modularization of code bases using templates (e.g., the Standard Template Library of C++).

Create: In [1] a transformation-based approach is presented in which scenario-oriented test cases can be generated from UML activity diagrams modeling concurrent applications. In the approach, transformation is used to translate activity diagrams to an intermediate representation having a tree-like structure. These tree structures are viewed as test specifications. Algorithms are then developed that derive tests cases corresponding to different scenarios and specific concurrent coverage criteria.

In previous work [14], we have explored the use of transformation to generate Java source code testing the class initialization (`<clinit>`) behavior of a targeted JVM. In this setting, an abstract model capturing the "first use" dependencies is given as input to a transformation-based test generator. The output produced is an executable Java program that exhaustively tests all prefix distinct clinit initialization sequences.

Check: Java PathExplorer (JPaX) is a runtime verification tool whose purpose is to monitor the execution of a Java program and check that the program satisfies a given set of properties expressed in temporal logic [6]. Within this system, the Maude rewriting system has been explored to implement finite trace semantics of (future time) linear temporal logic formulas[7]. This implementation enables JPaX to check that finite execution traces of a program satisfy various LTL properties, thereby enhancing the monitoring capabilities of JPaX.

VI. Summary and Conclusion

In this paper, an argument was presented that the type resolution function for Java is nontrivial, but also essential for source code analysis and manipulation tools. Section II gave an overview of some of central concepts that make type resolution, in Java, difficult. Sources of complexity include: (1) nested types, (2) static imports, and (3) the interplay between inheritance-based access control and nested types. A transformation-based approach was then presented in which the occurrence of a type reference anywhere within the subject source code triggers the generation of a *corresponding field declaration* (CFD). All field declarations are then decorated, via another transformation, with annotations containing the resolvant produced by an in-house resolution function. A special function containing "reflective hooks" is then added to each compilation unit. The resulting instrumentation enables a driver method to (1) compile the compilation units of the subject source code, (2) load the resulting class files using a class loader instance, and (3) certify that the information contained in type resolution annotations is identical to the type information obtained through reflection.

This approach can be effectively used to automatically certify the behavior of a type resolution function with respect to an arbitrary source code base. Of particular interest is the certification of the given type resolution function against a test suite exercising the numerous corner cases. However, in the absence of a truly exhaustive test suite, it is also of considerable interest to certify the type resolution function against an arbitrary code base such as an open source project.

Acknowledgment

This work was in part supported by the United States Department of Energy under Contract DE-AC04-94AL85000. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy.

References

[1] C. ai Sun. A Transformation-Based Approach to Generating Scenario-Oriented Test Cases from UML Activity Diagrams for Concurrent Applications. In *Computer Software and Applications, 2008. COMPSAC '08. 32nd Annual IEEE International*, pages 160 –167, 28 2008-aug. 1 2008.

[2] I. Baxter. Branch Coverage for Aribtrary Languages Made Easy. Technical Report DMS-2002, Semantic Designs, 2002.

[3] I. D. Baxter, C. Pidgeon, and M. Mehlich. DMS: Program Transformations for Practical Scalable Software Evolution. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 625–634, Washington, DC, USA, 2004. IEEE Computer Society.

[4] D. Flanagan. *Java In A Nutshell, 5th Edition*. O'Reilly Media, Inc., 2005.

[5] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. *The Java Language Specification (Java SE 7 Edition)*. Oracle, 2011.

[6] K. Havelund and G. Roşu. An Overview of the Runtime Verification Tool Java PathExplorer. *Form. Methods Syst. Des.*, 24(2):189–215, Mar. 2004.

[7] K. Havelund and G. Rosu. Monitoring Programs Using Rewriting. In *Proceedings of the 16th IEEE international conference on Automated software engineering*, ASE '01, pages 135–, Washington, DC, USA, 2001. IEEE Computer Society.

[8] M. Martins and A. Rosa. A Fault Injection Approach Based on Reflective Programming. In *Dependable Systems and Networks, 2000. DSN 2000. Proceedings International Conference on*, pages 407 –416, 2000.

[9] J. McCaffrey. Reflection-Based UI Testing. In *.NET Test Automation Recipes*, pages 33–63. Apress, 2006.

[10] J. T. Perry, V. Winter, H. Siy, S. Srinivasan, B. D. Farkas, and J. A. McCoy. The Difficulties of Type Resolution Algorithms. Technical Report SAND2010-8745, Sandia National Laboratories, December 2010.

[11] V. K. Proulx and W. Jossey. Unit test support for Java via reflection and annotations. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 49–56, New York, NY, USA, 2009. ACM.

[12] S. Roychoudhury, J. Gray, J. Zhang, P. Bangalore, and A. Skjellum. A Program Transformation Technique to Support AOP within C++ Templates. *Journal of Object Technology*, 9(1):143–160, Jan. 2010.

[13] The TL System, 2010.

[14] V. Winter. Model-driven Transformation-based Generation of Java Stress Tests. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 174(1):99–114, April 2007.

[15] V. Winter, J. Guerrero, A. James, and C. Reinke. Linking Syntactic and Semantic Models of Java Source Code within a Program Transformation System. In *Proceedings of the* 14^{th} *IEEE International Symposium on High Assurance Systems Engineering (HASE)*, 2012.

[16] V. L. Winter. Stack-based Strategic Control. In *Preproceedings of the Seventh International Workshop on Reduction Strategies in Rewriting and Programming*, June 2007.