Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2010, Article ID 540547, 7 pages
doi:10.1155/2010/540547

Research Article

AnnaBot: A Static Verifier for Java Annotation Usage

Ian Darwin

8748 10 Sideroad Adjala, RR 1, Palgrave, ON, Canada L0N 1P0

Correspondence should be addressed to Ian Darwin, ian@darwinsys.com

Received 16 June 2009; Accepted 9 November 2009

Academic Editor: Phillip Laplante

This paper describes AnnaBot, one of the first tools to verify correct use of Annotation-based metadata in the Java programming language. These Annotations are a standard Java 5 mechanism used to attach metadata to types, methods, or fields without using an external configuration file. A binary representation of the Annotation becomes part of the compiled ".class" file, for inspection by another component or library at runtime. Java Annotations were introduced into the Java language in 2004 and have become widely used in recent years due to their introduction in the Java Enterprise Edition 5, the Hibernate object-relational mapping API, the Spring Framework, and elsewhere. Despite that, mainstream development tools have not yet produced a widely-used verification tool to confirm correct configuration and placement of annotations external to the particular runtime component. While most of the examples in this paper use the Java Persistence API, AnnaBot is capable of verifying anyannotation-based API for which "claims"—description of annotation usage—are available. These claims can be written in Java or using a proposed Domain-Specific Language, which has been designed and a parser (but not the code generator) have been written.

1. Introduction

1.1. Java Annotations. Java Annotations were introduced into the language in 2004 [1] and have become widely used in recent years, especially since their introduction in the Java Enterprise Edition. Many open source projects including the Spring [2] and Seam [1] Frameworks, and the Hibernate and Toplink ORMs use annotations. So do many new Sun Java standards, including the Java Standard Edition, the Java Persistence API (an Object Relational Mapping API), the EJB container, and the Java API for XML Web Services (JAX-WS). Until now there has not been a general-purpose tool for independently verifying correct use of these annotations.

The syntax of Java Annotations is slightly unusual—while they are given class-style names (names begin with a capital letter by convention) and are stored in binary .class files, they may not be instantiated using the new operator. Instead, they are placed by themselves in the source code, preceding the element that is to be annotated (see Figure 1). Annotations may be compile-time or run-time; the latter's binary representation becomes part of the compiled "class" file, for inspection by another component at run time. Annotations are used by preceding their name with the "at" sign (@). For example, here is a class with both a compiletime annotation and a runtime annotation

The @WebService annotation from the Java JAX-WS API is a runtime annotation, used to tell a web container that this class should be exposed on the network as a SOAP-based web service. The @Override from Java SE is a compile-time annotation used to tell the compiler or IDE to ensure that the method is in fact overriding a method in the parent class.

1.2. Assertions. An assertion is a claim about the state of a computer program. Assertions have been used for many years in software verification. Goldschlager [4] devotes a section in his chapter on Algorithms to this topic. Voas et al. [5] describes a syntax for adding assertions to existing programs. A simple runtime assertion syntax was introduced using a new language keyword in version 1.4 of the Java programming language [6]. Assertion Languages have been used in verifying hardware designs [7].

1.3. Overview. This research describes a new tool for making assertions about how annotations are to be used, the Annotation Assertion-Based Object Testing tool or AnnaBot.

```
@WebService
public class Fred extends Caveman {
        @Override
        public void callHome() {
                // call Wilma here
        }
}
```

Figure 1: Java SE and EE Annotations applied to a SOAP Web Service class.

```
@Entity public class Person {
  @Id int id;
  @Column(name="given_name")
  public String getFirstName() {...}
```

Figure 2: Portion of a flawed JPA Entity Class.

Section 2 describes the origins of the research in an otherwise-undetected failure and describes how the resulting tool operates in general terms, giving some examples of the classes of errors that it can detect. Section 3 discusses the internal operation of the verification tool and the two methods of writing "claims": Java code and the planned DSL.

Section 4 describes the experiences in using this verification tool on a couple of sample code bases, one derived from the author's consulting practice and one from the examples provided with a popular web development framework. Section 5 details some additional research and development that should be undertaken in the future, along lines primarily suggested by Section 4.

Section 6 discusses Related Research, primarily in the area of annotation verification. Section 7 provides details on obtaining the verification tool for anyone who would like to replicate the results, apply it in different domains, or provide improvements to the software.

2. AnnaBot: The Annotation Assertion Verification Tool

Like most good tools, AnnaBot has its origin in a real-life problem. Sun's Java Enterprise Edition features the Java Persistence Architecture or JPA [8]. The JPA API is designed to facilitate mapping Java objects into relational database tables. JPA is heavily biased towards use of Java5 annotations to attach metadata describing how classes are mapped to and from the relational database. The `@Entity` annotation on a class marks the class as persistent data. Each persistent entity class must have an `@Id` annotation marking the field(s) that constitutes the primary key in the database. Other properties may be annotated (on the field or on the get method) with other annotations such as `@Column` if the database column name must differ from the Java property name—for example, a Java field called *firstName* might map to the SQL column `GIVEN_NAME`. A brief look at the documentation might lead one to write Figure 2.

The code in Figure 2 may fail mysteriously, because the specification states that one can only annotate fields *or* getters. The code as shown will compile correctly, but when deployed and run, the misleading error message you get—if any—depends on whether you are using Hibernate, EclipseLink, or some other Provider behind JPA. Having wasted half a day trying to track down this error, the author concluded that others might make the same mistake (or similar errors, such as annotating the setter method instead of the getter, which are generally silently ignored).

Yet it is not just my one error that makes a tool such as Annabot useful. Consider the scenario of a JPA- or Hibernate-based enterprise software project undergoing development and maintenance over several years. While the odds of an experienced developer annotating a field instead of a method in JPA (or similar error in the other APIs) are small, it could happen due to carelessness. Add in a couple of junior programmers and some tight deadlines, and the odds increase. The consequences of such a mistake range from almost none—since JPA does tend to provide sensible defaults—to considerable time wasted debugging, what appears to be a correct annotation, the ability to verify externally that the annotations have been used correctly, especially considering the almost zero cost of doing so, makes it very much worthwhile.

And, above all, AnnaBot is not limited to JPA. It can—with provision of suitable metadata files—be used to verify correct annotation use in *any* API that uses Java 5 annotations for runtime discovery of metadata.

The tool works by first reading one or more "claim" files (the term `claim` was chosen because `assert` is already a keyword in Java), which claim or assert certain valid conditions about the API(s) being used. It is anticipated that the tool will gradually acquire a library of "standard" claim files for the more common APIs (both client- and server-side) which are using Annotations. For example, Figure 3 is an example of a partial Assertion file for verifying correct use of annotations in the Java Persistence API. The JPA specification [8, Section 2.1.1] states that annotations may be placed before fields (in which case "field access" via Reflection is used to load/store the properties from/to the object) or before the get methods (in which case these methods and the corresponding set methods are used). It notes that "the behavior is unspecified if mapping annotations are applied to both persistent fields and properties or if the XML descriptor specifies use of different access types within a class hierarchy."

As can be seen in Figure 3, the syntax of a claim file is reminiscent of Java; this is intentional. The `import` statement, for example, has the same syntax and semantics as the like-named statement in Java; it makes annotations available by their unqualified name. The first `if` statement says that any class decorated with the `@Entity` annotation must have a method or field annotated with the `@Id` annotation—"data to be persisted must have a primary key," in database terms. The methods *field.annotated()* and *method.annotated()* can be called either with a single class name or, as is done here, with a package wildcard with similar syntax and semantics as on Java's `import` statement (but

```
import javax.persistence.Entity;
import javax.persistence.Id;

claim JPA {
if (class.annotated(javax.persistence.Entity)) {
    require method.annotated(javax.persistence.Id)
        || field.annotated(javax.persistence.Id);
  atMostOne     method.annotated(javax.persistence.ANY)
||      field.annotated(javax.persistence.ANY)
        error "The JPA Spec only allows JPA annotations on methods OR fields";
};
if (class.annotated(javax.persistence.Embeddable)) {
        noneof   method.annotated(javax.persistence.Id) ||
        field.annotated(javax.persistence.Id);
};
}
```

Figure 3: JPA Claim in AnnaBot DSL.

```
claim EJB3Type {
  atMostOne
   class.annotated(javax.ejb.Stateless),

   class.annotated(javax.ejb.Stateful)
      error "Class has conflicting top-level EJB
annotations"
    ;
}
```

Figure 4: EJB Claim file in AnnaBot DSL.

```
public interface PrePostVerify {
       void preVerify();
       void postVerify();
}
```

Figure 5: PrePostVerify interface.

with ANY instead due to an apparent issue with the parser generator). The example also claims that you must not annotate both methods and fields with JPA annotations.

The second `if` statement says that `Embeddable` data items (items which share the primary key and the row storage of a "main" `Entity`) must *not* have an `@Id`—Embeddable objects are not allowed to have a different primary key from the row they are contained in.

As a second example, here is a rudimentary claim file for one aspect of the Enterprise JavaBean (EJB Version 3.0) specification [8].

The example in Figure 4 shows use of `atMostOne` at class level to confirm that no class is annotated with more than one of the mutually-exclusive Annotation types listed (`Stateless` or `Stateful`—these are "Session Beans"). Beside these and Entity, there are other types of EJB that are less common—the example is not intended to be complete or comprehensive.

2.1. Cross-Class Tests. There are some tests that cannot be performed merely by examining a single class. To draw another example from the Java Persistence API, the choice between field annotation and accessor annotation must be consistent not only within a class (as the JPA Claim above tests) but also across all the classes loaded into a given "persistence unit"—usually meaning all the JPA entity classes in an application. The Claim Files shown above cannot handle this. Annabot has recently been modified to support Java-based Claim classes having an optional `implements PrePostVerify` clause. The `PrePostVerify` interface shown in Figure 5 could be used, for example, to allow the claim to set booleans during the claim testing phase and examine them in the `postVerify` method, called as its name suggests after all that the claim verification has been completed.

3. Implementation

The basic operation of Annabot's use of the reflection API is shown in class AnnaBot0.java in Figure 6. This demonstration has no configuration input; it simply hard-codes a single claim about the Java Persistence Architecture API, that only methods *or* only fields be JPA-annotated. This version was a small, simple proof-of-concept and did one thing well.

The Java class under investigation is accessed using Java's built-in Reflection API [9]. There are other reflection-like packages for Java such as Javassist [10] and the Apache Software Foundation's Byte Code Engineering Language [11]. Use of the standard API avoids dependencies on external APIs and avoids both the original author and potential contributors having to learn an additional API. Figure 6 is a portion of the code from the original `AnnaBot0` which determines whether the class under test contains any fields with JPA annotations.

To make the program generally useful, it was necessary to introduce some flexibility into the processing. It was decided to design and implement a Domain-Specific Language [12, 13] to allow declarative rather than procedural specification of additional checking. One will still be able to extend the functionality of AnnaBot using Java, but some will find it more convenient to use the DSL.

The first version of the language uses the Java compiler to convert claim files into a runnable form. Thus, it is slightly

```
Field[] fields = c.getDeclaredFields();
boolean fieldHasJpaAnno = false;
for (Field field : fields) {
        Annotation[] ann =
        field.getDeclaredAnnotations();
        for (Annotation a : ann) {
                Package pkg =
                        a.annotationType().
                        getPackage();
                if (pkg != null &&
                        pkg.getName().
                        startsWith(
                        "javax.persistence"))
                {
                        fieldHasJpaAnno =
                                true;
                        break;
                }
        }
}

// Similar code for checking if any
// JPA annotations are found on methods

// Then the test
if (fieldHasJpaAnno &&
    methodHasJpaAnno) {
    error("JPA Annotations should be on methods or
fields, not both");
}
```

Figure 6: Portion of AnnaBot0.java.

```
package jpa;

import annabot.Claim;
import tree.*;

public class JPAEntityMethodFieldClaim extends Claim
{
        public String getDescription() {
                return "JPA Entities may have field
OR method annotations, not both";
        }
        public Operator[] getClassFilter() {
                return new Operator[] {
                new ClassAnnotated(
                "javax.persistence.Entity"),
                };
        }

        public Operator[] getOperators() {
                return new Operator[] {
                    new AtMostOne(
                    new FieldAnnotated(
                        "javax.persistence.*"),
                    new MethodAnnotated(
                        "javax.persistence.*",))
                };
        }
}
```

Figure 7: JPA Claim written in Java.

more verbose than the Annabot Language. Figure 7 is the JPA claim from AnnaBot0 rewritten as a Java-based Claim.

While it is more verbose than the DSL version, it is still almost entirely declarative. Further, as can be seen, there is a fairly simple translation between the Java declarative form and the DSL. The AnnaBotC compiler tool will translate claim files from the DSL into Java class files.

```
program:      import_stmt*
        CLAIM IDENTIFIER '{'
              stmt+
        '}'
        ;

import_stmt:  IMPORT NAMEINPACKAGE ';'
        ;

// Statement, with or without an
// if ... { stmt } around.
stmt:   IF '(' checks ')' '{' phrase+ '}' ';'
        | phrase
        ;
phrase: verb checks error? ';'
        ;

verb:   REQUIRE | ATMOSTONE | NONEOF;

checks: check
                | NOT check
                | ( check OR check )
                | ( check AND check )
                | ( check ',' check)
                ;

check:  classAnnotated
        | methodAnnotated
        | fieldAnnotated;

classAnnotated: CLASS_ANNOTATED '('
         NAMEINPACKAGE ')';
methodAnnotated:        METHOD_ANNOTATED '('
         NAMEINPACKAGE ')';
fieldAnnotated: FIELD_ANNOTATED '('
         NAMEINPACKAGE
        ( ',' MEMBERNAME )? ')'
        ;

error:  ERROR QSTRING;
```

Figure 8: EBNF for DSL.

The structure of the DSL suggested that an LL or an LR parser would be adequate. While any type of parser may in theory be written by hand, software developers have used parser generator tools for three decades, starting with the widely-known UNIX tool YACC [14]. There are many "parser generator" tools available for the Java developer; a collection of them is maintained on the web [15]. After some evaluation of the various parser generators, a decision was made to use Antlr [16]. This was based in part on Parr's enthusiasm for his compiler tool, and his comments in [16] about LR versus LL parsing (YACC is LR, Antlr is LL). "In contrast, LL recognizers are goal-oriented. They start with a rule in mind and then try to match the alternatives. For this reason, LL is easier for humans to understand because it mirrors our own innate language recognition mechanism..."

A basic EBNF description of the AnnaBot input language is in Figure 8; the lexical tokens (names in upper case) have been omitted as they are obvious. Figures 3 and 4 provide examples of the DSL.

This provides sufficient structure for the current design of the DSL to be matched. The parser has been fully implemented in Antlr, but the code generation is not yet implemented. For the present, claims files are written in Java (as in Figure 7).

Table 1: Usage results.

Codebase	Kloc	Classes Entity/Total	Errors	Time (Seconds)
seambay	2.27	7, 22	0	0.6
TCP	26.8	94/156	0	~3 (see Section 4.1)

4. Experiences and Evaluation

The current version has been used to verify a medium-scale web site that is currently being redeveloped using the Seam framework; Seam uses JPA and, optionally, EJB components. The site, which consists of approximately one hundred JPA "entity" classes and a growing number of controller classes, is being developed at the Toronto Centre for Phenogenomics, or TCP [17]. Although AnnaBot has not yet found any actual claim violations in the TCP software—many of the Entity classes were generated automatically by the Hibernate tooling provided for this purpose—AnnaBot provides an ongoing verification that no incorrect annotation uses are being added during development. With a small starter set of four JPA and EJB claims in service, the process takes a few seconds to verify about a hundred and fifty class files. It thus has a very low cost for the potential gain of finding a column that might not get mapped correctly or a constraint that might be violated at runtime in the database.

As a second example, the program was tested against the `seambay` package from the Seam Framework [3] examples folder. As the name implies, `seambay` is an auction site, a small-scale imitation of the well-known eBay.com site. Seam version 2.2.0.GA provides a version of `seambay` that has only 20–30 Java classes depending on version, perhaps to show how powerful Seam itself is as a framework. AnnaBot scanned these in 0.6 seconds with four JPA claims, and again unsurprisingly found no errors (Table 1).

The result of these tests indicate that AnnaBot is a very low-overhead and functioning method of verifying correct use of annotations.

4.1. A Note on Performance. The one risk as the program's use grows is the $O(m\,n)$ running time for the testing phase—for m tests against n target classes, the program must obviously perform $m \times n$ individual tests. To measure this, I created four additional copies of the JPA Entity claim, to double the value of m, and re-ran AnnaBot on the TCP corpus of ~150 classes. Running time for the initialization phase (discovering and loading classes) actually went down slightly; running time for the testing phase went from 2.3 seconds with 4 claim files to 3.2 seconds with 8. All times were measured using Java's `System.currentTimeMillis()` on a basically otherwise idle laptop (An AMD Athlon-64, 3500+, 1800 MHz single core, 2 GB of RAM, running OpenBSD 4.4 with the X Window System and KDE; AnnaBot running in Eclipse 3.2 using Java JDK 1.6.0), and averaged over 3 or more runs. These results would indicate that AnnaBot's performance scales reasonably when running tests. Therefore, adding a useful library of claim files should not create a significant obstacle to running the tests periodically.

5. Future Development

As stated above, experience has shown this to be a low-cost verification method for software. The following enhancements are under consideration for the tool, subject to time and availability.

5.1. Finish Implementation of the Compiler. The parser is written and tested, and the DSL-based claims in this paper have been parsed using the version current as of November, 2009. The parser needs actions added to generate the class; it is planned to use the Javassist API [10] to generate the .class file corresponding to the claim file being read.

5.2. Convert from Java Reflection to Javassist. Given the use of Javassist in the compiler, it may be wise to convert the inspection code in AnnaBot to use Javassist in the verification as well. Java's built-in reflection works well enough, but it requires that all annotation classes be available on the CLASSPATH. This can become an irritation in using AnnaBot; using Javassist to do the code inspection should eliminate this.

5.3. More Fine-Grained Verification. At present only the presence or absence of a given annotation can be verified. For more complete verification, it ought to be possible to interrogate the attributes within an Annotation, for example,

```
@Entity @Table(name="this is not a
valid table name")
```

This would, like most of the examples in this paper, be considered acceptable at compile time (the *name* element merely requires a Java String as its value), but would fail at run time, since "this is not a valid table name" is indeed not a valid table name in the SQL language of most relational databases.

Some consideration has been given to a syntax for this, probably using regular expressions similar to what is done for method names, but nothing concrete has been established.

5.4. Run as Plug-in. The Eclipse IDE [18] and the FindBugs static testing tool [19] are both widely used tools, and both provide extensibility via plug-ins. However, FindBugs uses its own mechanism for parsing the binary class files (for an example, see Listing 4 of Goetz [20]). It may or may not be feasible to reconcile these differing methods of reading class files to make AnnaBot usable as a FindBugs plug-in. Failing this, it would definitely be worth while to make AnnaBot usable as an Eclipse plug-in, given the wide adoption of Eclipse in the Java developer community.

6. Related Research

Relatively little attention has been paid to developing tools that assist in verifying correct use of annotations. Eichberg et al. [21] produced an Eclipse plug-in which uses a different, and I think rather clever, approach: they preprocess

the Java class into an XML representation of the byte-code, then use XPath to query for correct use of annotations. This allows them to verify some nonannotation-related attributes of the software's specification conformance. For example, they can check that EJB classes have a no-argument constructor (which Annabot can easily do at present). They can also verify that such a class does not create new threads. Annabot cannot do this at present since that analysis requires inspection of the bytecode to check for "monitor locking" machine instructions. However, this is outside my research's scope of verifying the correct use of annotations. It could be implemented by using one of the non-Sun "reflection" APIs mentioned in Section 3.

The downside of Eichberg's approach is that all classes in the target system must be preprocessed, whereas AnnaBot simply examines target classes by reflection, making it faster and simpler to use.

Noguera and Pawlak [22] explore an alternate approach. They produce a rather powerful annotation verifier called AVal. However, as they point out, "AVal follows the idea that annotations should describe the way in which they should be validated, and that self validation is expressed by meta-annotations (@Validators)." Since my research goal was to explore validation of existing annotation-based APIs provided by Sun, SpringSource, Hibernate project, and others, I did not pursue investigation of procedures that would have required attempting to convince each API provider to modify their annotations.

JavaCOP by Andreae [23] provides a very comprehensive type checking system for Java programs; it provides several forms of type checking, but goes beyond the use of annotations to provide a complete constraint system.

The work of JSR-305 [24] has been suggested as relevant. JSR-305 is more concerned with specifying new annotations for making assertions about standard Java than with ensuring correct use of annotations in code written to more specialized APIs. As the project describes itself, "This JSR will work to develop standard annotations (such as @NonNull) that can be applied to Java programs to assist tools that detect software defects."

Similarly, the work of JSR-308 [25], an offshoot of JSR-305, has been suggested as relevant, but it is concerned with altering the syntax of Java itself to extend the number of places where annotations are allowed. For example, it would be convenient if annotations could be applied to Java 5+ Type Parameters. This is not allowed by the compilers at present but will be when JSR-308 becomes part of the language, possibly as early as Java 7.

Neither JSR-305 nor JSR-308 provides any support for finding misplaced annotations.

7. Where to Obtain the Software

The home page on the web for the project is http://www.darwinsys.com/annabot/.

The source code can be obtained by Anonymous CVS from the author's server, using these command-line tools or their equivalent in an IDE or other tool:

```
export CVSROOT=:pserver:\

anoncvs@cvs.darwinsys.com:/cvspublic

cvs checkout annabot
```

Contributions of patches or new Claim files will be most gratefully received.

Acknowledgments

Research performed in partial fulfillment of the M.Sc degree at Staffordshire University. The web site development which led to the idea for AnnaBot was done while under contract to The Toronto Centre for Phenogenomics [17]; however, the software was developed and tested on the author's own time. Ben Rady suggested the use of Javassist in the Compiler. Several anonymous reviewers contributed significantly to the readability and accuracy of this paper.

References

[1] Java 5 Annotations, November 2009, http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html.

[2] Spring Framework Home Page, October 2009, http://www.springsource.org/.

[3] G. King, "Seam Web/JavaEE Framework," October 2009, http://www.seamframework.org/.

[4] L. Goldschlager, *Computer Science: A Modern Introduction*, Prentice-Hall, Upper Saddle River, NJ, USA, 1992.

[5] J. Voas, et al., "A Testability-based Assertion Placement Tool for Object-Oriented Software," October 1997, http://hissa.nist.gov/latex/htmlver.html.

[6] "Java Programming with Assertions," November 2009, http://java.sun.com/j2se/1.4.2/docs/guide/lang/assert.html.

[7] J. A. Darringer, "The application of program verification techniques to hardware verification," in *Proceedings of the Annual ACM IEEE Design Automation Conference*, pp. 373–379, ACM, 1988.

[8] "EJB3 and JPA Specifications," November 2009, http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html.

[9] I. Darwin, "The reflection API," in *Java Cookbook*, chapter 25, O'Reilly, Sebastopol, Calif, USA, 2004.

[10] Javassist bytecode manipulation library, November 2009, http://www.csg.is.titech.ac.jp/~chiba/javassist/.

[11] Apache BCEL—Byte Code Engineering Library, November 2009, http://jakarta.apache.org/bcel/.

[12] J. Bentley, "Programming pearls: little languages," *Communications of the ACM*, vol. 29, no. 8, pp. 711–721, 1986.

[13] I. Darwin, "PageUnit: A "Little Language" for Testing Web Applications," Staffordshire University report, 2006, http://www.pageunit.org/.

[14] S. Johnson, "YACC: yet another compiler-compiler," Tech. Rep. CSTR-32, Bell Laboratories, Madison, Wis, USA, 1978.

[15] "Open Source Parser Generators in Java," April 2009, http://java-source.net/open-source/parser-generators.

[16] T. Parr, *The Definitive ANTLR Reference: Building Domain-Specific Languages*, Pragmatic Bookshelf, Raleigh, NC, USA, 2007.

[17] Toronto Centre for Phenogenomics, April 2009, http://www.phenogenomics.ca/.

[18] Eclipse Foundation, Eclipse IDE project, November 2009, http://www.eclipse.org/.

[19] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *ACM SIGPLAN Notices*, vol. 39, no. 12, pp. 92–106, 2004.
[20] B. Goetz, "Java theory and practice: testing with leverage—part 1," April 2009, http://www.ibm.com/developerworks/library/j-jtp06206.html.
[21] M. Eichberg, T. Schäfer, and M. Mezini, "Using annotations to check structural properties of classes," in *Proceedings of the 8th International Conference on Fundamental Approaches to Software Engineering (FASE '05)*, pp. 237–252, Edinburgh, UK, April 2005.
[22] C. Noguera and R. Pawlak, "AVal: an extensible attribute-oriented programming validator for Java," in *Proceedings of the 6th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM '06)*, pp. 175–183, Philadelphia, Pa, USA, September 2006.
[23] C. Andreae, "JavaCOP—User-defined Constraints on Java Programs," November 2009, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.96.3014&rep=rep1&type=pdf.
[24] JSR-305, November 2009, http://jcp.org/en/jsr/detail?id=305.
[25] JSR-308, November 2009, http://jcp.org/en/jsr/detail?id=308.