Cross-Tests and Composition in Annotation-Based Content-Validation Frameworks

Blinded

Abstract

In this paper we investigate the use of Java annotations for software security purposes. In particular, we implement a framework for content validation, with focus on input validation. The tests are specified by annotations, which are then retrieved by reflection and executed. This eliminates the need for external XML configuration files, and separates the testing code from the application code, by allowing easy creation and reuse of custom tests. In our framework we extend the type of tests that can be defined through annotations. We provide the possibility of creating tests involving multiple values. The flexibility and reusability of tests are also improved by allowing composition and boolean expressions.

1 Introduction

The OWASP Top Ten Project [1] lists the lack of proper input validation as the most prevalent cause of critical software vulnerabilities. For this reason, it is important to check that all input satisfies the criteria under which it is safe to execute the program. As an example, take a Java program performing integer division. Integer division by 0 is an illegal operation, resulting in a runtime exception. Hence the value of the divisor should always be validated.

Carefully designing the application can alleviate problems caused by incorrect input. However, this alone will not prevent problems that might arise when a bad input is either passed on to other subsystems like databases, or manipulated and returned to the user as harmful code.

Standard *input validation* mechanisms should make sure that all input is validated for length, type, syntax, and business rules before accepting the data to be displayed, stored or used [1]. This task can be repetitive and tedious for a programmer, and this is is the primary motive for implementing frameworks for input validation (Common Validator [2], Struts 2 [3], Hibernate [4] and Heimdall [5]). Such frameworks make it easier to maintain and execute the testing code by decoupling the application logic from the validation logic.

For object-oriented languages like Java, the challenge is to validate specific properties of an object representing the input, without writing validation code in the object itself. Historically, XML configuration files have been used to achieve this separation of concerns, by explicitly storing the names of the properties to be tested and that of the tests to be performed. At runtime, *reflection* [6] or Servlet filters (listener or interceptors) [3] would then be used to actually run the tests on the target methods.

An alternate solution, based on *annotations*, which were introduced in Java 5.0 [6], has gradually emerged. Approaches for input validation based on this new technology are described in [7, 8, 9], and employed, for instance, by Struts 2 [3] and Hibernate [4].

Our approach is inspired by Heimdall [5], but adopts annotations instead of XML configuration. The reasons to prefer annotations over XML configuration files have been motivated in [8], and here we explore how far annotations can be pushed for input validation purposes. Although some technical solutions we use are also

found in [7, 8, 9], we offer a simpler and more powerful way of creating custom tests, with focus on reusability. Furthermore, we propose a way of extending validation over multiple properties of an object, rather than a single property. For this purpose we distinguish between *property-tests* and *cross-tests*. A property-test is used for the validation of a single object property (for instance, JavaBean properties accessible through getter-methods), whereas a cross-test is concerned with constraints involving multiple properties.

The next section shows a simple example of how annotations can be used for validation. This running example is gradually extended to show more advanced features of our framework. A more formal description of how new annotations can be created and used is given in Section 3. The implementation details are explained through sequence and object diagrams in Section 4. Finally we compare our work to the other framework we mentioned previously in this section, and draw some conclusions..

2 Working Example

In this section we introduce the working example used throughout the paper, and show how annotations can be used to define tests on single properties of an object.

We will use the web form for international money transfers from a hypothetical Internet bank (see Figure 1). *IBAN* (International Bank Account Number) is the standard for identifying bank accounts internationally. Some countries did not adopt this standard, and for money transfer to these countries, a special *clearing code* is needed in combination with the normal account number of the beneficiary. *BIC* (Bank Identifier Code), also known as SWIFT. It is needed to identify the beneficiary's bank uniquely.

We assume that the object representing the form is created in Java, and that each field in the web form is represented by a property of this object. Fields where the user does not enter a value, are in this example represented by the `null` value. A partial implementation of this Java object is shown in Figure 2. Here every annotation represents a test to be run on the return value of the method it is applied to. In our framework, annotations representing tests are called *validation-annotations*. This categorization is further split into *property-annotations*, which represent property-tests, and *cross-annotations*, which represent cross-tests. All the annotations in Figure 2 are property-annotations, i.e., they involve checking a single specific property.

We use property-tests to check whether basic formatting rules are respected. For example, the annotation `@IntRange(min=0,max=10000)`

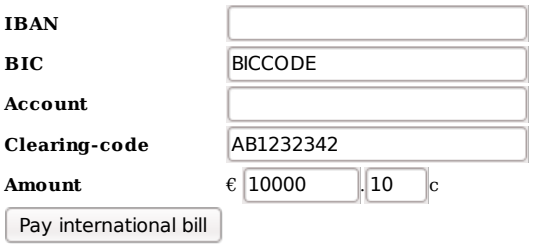


Figure 1: *Web form for international bank transfer.*

```
@ValidateBIC
@Required
public String getBIC() {
 return BIC;
}

@IntRange(min=0,max=10000)
public Integer getAmountEuro() {
 return amountEuro;
}

@IntRange(min=0,max=99)
public Integer getAmountCents() {
 return amountCents;
}
```

Figure 2: *Example code using the property-annotations to test input from the web form in Figure 1.*

represents a test that checks whether the value of `amountEuro` is non-negative and not greater than 10000. The property-annotation `@IntRange(min=0,max=99)` represents a test to check whether `amountCents` is between 0 and 99. The property-annotation `@ValidateBIC` represents a property-test for BIC codes, and `@Required` means that the field cannot be left empty.

The annotations themselves only specify what tests should be run on each value. To actually run the tests, an object must be passed to a validator. The validator inspects the object through reflection, extracts the annotations and the return values from the getter-methods, and invokes the corresponding tests. This process is discussed in details in Section 4.

3 Validation annotations and tests

In this section we discuss the reasons for distinguishing between property-tests and cross-tests, and provide details of how they are implemented and used.

3.1 Property-annotations and Property-tests

Creating a property-annotation is fairly straightforward. As an example we use the declaration of `@IntRange` shown in Figure 3.

A fundamental part of the declaration is the meta-annotation `@Validation`, which works as a marker. Without it, our framework would not be able to distinguish a property-annotation from other annotations. There are other solutions to this problem, but such *marker-annotation*s are a standard way to compensate for the lack of inheritance in annotations [8, 7, 9].

The `@Retention(retentionPolicy.RUNTIME)` meta-annotation must be present such that the property-annotation is accessible at runtime. The annotation `@Target` has the usual meaning, but we will explain in the next sections why we use also `ElementType.ANNOTATION_TYPE` besides `ElementType.METHOD`. The annotation declaration itself is fairly standard and can be annotated with any number of other annotations.

```
@Validation
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.ANNOTATION_TYPE,
         ElementType.METHOD})
public @interface IntRange {
 int min();
 int max();
 public static class Tester
 implements IPropertyTester{
  public boolean runTest(IntRange r,
                         Integer v) {
   return(v >= r.min() && v <= r.max());
  }
 }
}
```

Figure 3: *Example of property-annotation.*

Finally, we require a public *inner class* which must contain the code of the property-test associated with this property-annotation. This class must implement the interface shown in Figure 4 in order to ensure that it provides the implementation of the method `runTest()`, which is invoked by the framework to run the test. Another possible approach for associating a test to an annotation is explained in [7].

The test corresponding to the property-annotation `@IntRange` is defined in the inner class `Tester`, as shown in Figure 3. The method `runTest()` is called by reflection and takes as parameters an instance of the annotation and the object to test (that is, the return value of the method). We allow only one inner class implementing `IPropertyTester` in the annotation declaration.

3.1.1 Handling `null` values

Many validation frameworks provide an annotation `@Required` which indicates that a certain property should not be `null` [3, 4, 5]. However,

```
public interface IPropertyTester
 <A extends Annotation,I> {
  public boolean runTest(A an, I o)
  throws ValidationException;
}
```

Figure 4: *The interface for the classes implementing property-tests.*

no annotation seems to be provided to specify when a property *can* be `null`.

To understand why this might be useful, let us assume that we allowed the field BIC in Figure 1 to be left empty by the user. This means that a `null` return value from the method `getBIC()` should not lead to a `NullPointerException` being thrown during the validation process. To achieve this, either the test represented by `@validateBIC` must be able to correctly handle a `null` value in this situation, or the framework should prevent any test to be run when `BIC` has the value `null`. In the first case the burden of treating this special case is left to the programmer, who must consider the possibility that any test he or she designs might be run on a `null` value. However, tests are supposed to be reusable and cannot account for all possible ways of treating a `null` value in different situations. The same problem arises when, in the absence of a `@Required` annotation, the framework should decide how to interpret a `null` value, by risking to mask a possible error or causing one.

To avoid these problems, we provide the `@NotRequired` annotation, which can be used to specify that a `null` return value is valid, and that in this case no other tests should be run on the value. If neither a `@Required` nor a `@NotRequired` annotation is specified, the framework will simply run the other tests on the method, even if the return value is `null`. Therefore, by default, our framework does not give any special treatment to `null` values, and the programmer can design reusable tests, by handling a `null` value in an independent way. In this setting, any `NullPointerException` will properly signal a programming error.

```
@Required
public String getBIC() { return BIC; }

@ExactlyOneNull
@NotRequired
public String getIBAN() { return IBAN; }

@ExactlyOneNull
@AllOrNoneNull
@NotRequired
public String getAccount() {
 return account;
}

@AllOrNoneNull
@NotRequired
public String getClearingcode() {
 return clearingcode;
}
```

Figure 5: *Examples of cross-annotations.*

3.2 Cross-annotations

Recall the specifications of international bank transfers mentioned in Section 2. All transfers require the BIC code of the receiving bank, and in addition either the IBAN or both clearing code and the account number. This means that there is a mutual dependence among some fields of the web form. Therefore, in order to check such constraints in the corresponding object, it is not enough to consider the return values of the involved methods independently. For this purpose we introduce a new type of validation-annotation which we have called cross-annotations. These allow a programmer to create tests involving multiple properties of an object, i.e., cross-tests.

In Figure 5 we extend the example in Figure 2 to show how it is possible to annotate the web form object in order to enforce the constraints mentioned earlier. Each cross-test is represented by a cross-annotation, which is applied to all methods whose return values are involved in the test. All annotations in the example are cross-annotations with the exception of `@Required` and `@NotRequired`. The property-annotations

from Figure 2 are not shown in order to keep the example readable.

The cross-test represented by `@ExactlyOneNull`, which is applied to the return values of `IBAN` and `clearingCode`, ensures that exactly one of them has not been filled in the web form. Furthermore, the cross-test represented by `@AllOrNoneNull` makes sure that either all or none of the methods marked with it return `null`. Thus, we are able to check that either the IBAN is used, or both the account number and the clearing-code are specified, but not all three.

Cross-annotations can be declared in almost the same way as property-annotations, as shown in Figure 6. Only the marker-annotation, `@CrossValidation`, and the interface of the inner test class, shown in Figure 7, are different.

```
@CrossValidation
public @interface AllOrNoneNull {
 public static class Tester implements
 ICrossTester<AllOrNoneNull,String> {

  public boolean runTest(AllOrNoneNull c,
                    ArrayList<String> v) {
   ...
 }
}
```

Figure 6: *Example of cross-annotation declarations.*

As can be seen in Figure 7, a cross-test takes as parameter the corresponding cross-annotation and all the return values involved in the test as a single `ArrayList`. This means that the return values are not differentiated according to what method they come from, hence limiting the type

```
public interface ICrossTester
<A extends Annotation, V> {
    public boolean runTest
       (A a, ArrayList<V> v)
    throws ValidationException;
}
```

Figure 7: *The interface for cross-tests.*

```
@Validation
@BoolTest(BoolType.AND)
@PatMatch("\\w{8}|\\w{11}")
@AdditionalTest
public @interface ValidateBIC{}
```

Figure 8: *Definition of the annotation* `@ValidateBIC`*, used in Figure 2.*

of cross-tests that can be developed in the current framework. These limitations are discussed in the next section.

3.3 Boolean composition

Another novelty of our approach is that we can combine validation-annotations with boolean operators in order to create new validation-annotations. These *composed* annotations can be created by declaring a new validation-annotation which is annotated with the validation-annotations we want to compose. In addition, the special meta-annotation `@BoolTest` can also be used in the composition. Its single element is of type `public enum BoolType{OR, AND, ALLFALSE}`, with the usual semantics. By default, specifying a list of annotations without the `@BoolTest` annotation represents the conjunction of the corresponding tests, thus `BoolType.AND` is not strictly necessary.

Figure 8 shows the declaration of the annotation `@ValidateBIC` which we first introduced in Figure 2. This annotation is created by composing `@PatMatch("\\w{8}|\\w{11}")`, which is a common annotation for string-matching tests, and the one represented by `@AdditionalTest`, which represents some other possible test that we do not specify here. Since the annotation `@BoolTest(BoolType.AND)` is also specified, the test represented by `@ValidateBIC` will succeed only if *both* the tests represented by the two other property-annotations succeed.

Boolean composition can also be applied with cross-annotations. For example, if we in the web form above want to check that the over-

```
@AmountCheck
public Integer getAmountEuro() {
 return amountEuro;
}

@AmountCheck
public Integer getAmountCents() {
 return amountCents;
}
```

Figure 9: *Examples of cross-annotations.*

```
// MaxAmount declaration
@CrossValidation
@BoolTest(BoolType.OR)
@OneLessThan(1)
@AllLessThan(10000)
public @interface MaxAmount {}

//AmountCheck declaration
@CrossValidation
@BoolTest(BoolType.AND)
@SumMin(1)
@MaxAmount
public @interface AmountCheck {}
```

Figure 10: *Composition of cross-annotations.*

all amount transferred is greater than 0.00, but not greater than 10000.00, we can use the cross-annotation `@AmountCheck` as shown in Figure 9, since the fields for Euros and cents are represented as different properties in the webform object.

In Figure 10, we see that `@AmountCheck` is a composition of two other cross-annotations: `@SumMin(1)` and `@MaxAmount`. The first represents a test checking that the sum of `amountEuro` and `amountCents` is greater than 0. The second is in turn a composed cross-annotation, which by checking that either one of the two values is smaller than 1, or both are smaller than 10000, can guarantee that the total amount they represent together is at most 10000. This example also shows that composition is recursive. This allows the encapsulation of a complicated validation policy into a single annotation, thus improving readability, usage, and reusability.

Table 1: *Examples of each of the four types of tests.*

	Basic Tests	Composed Tests
Property-Tests	`@IntRange`	`@ValidateBIC`
Cross-Tests	`@AllOrNoneNull`	`@AmountCheck`

Now that we introduced composition, it might become clearer why we imposed the limitations discussed in the previous section on cross-tests. If we allowed elements in cross-annotations which could be associated to a particular return value, for instance to identify the method they came from, composition would become more involved.

However, elements that are used to configure the test represented by the annotation, as in `@SumMin`, are allowed. Furthermore, since this parameter should be the same for each instance of the cross-annotation appearing in the object, it is a good practice to encapsulate the annotation with the specific element value into a new cross-annotation without any elements.

Using composition to encapsulate an annotation with specific argument values into one without parameters, can also ease maintenance or/and refactoring. Also, boolean composition makes it particularly easy to create validation tests based on white- and blacklisting.

4 Validation Process

In this section we present more implementation details of the framework. We assume that all properties of an object are available through no-arg methods, e.g. getter methods in JavaBeans. This is not an important restriction, especially for objects representing an input. If existing classes allow access to input only through methods with parameters, it should be easy to add a no-arg method, which then supplies default argument(s) to the former method.

The methods returning values of properties to be tested must be annotated by the programmer

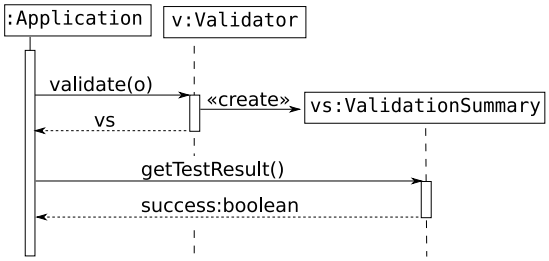


Figure 11: *Sequence diagram for the main steps of the validation process, where* `o` *is the object to be tested.*

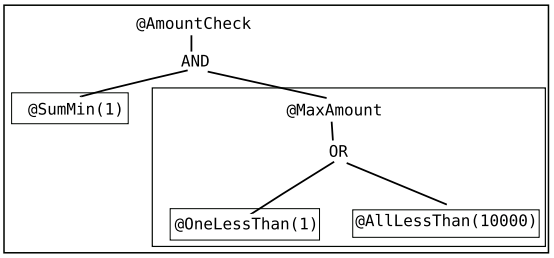


Figure 13: *Diagram showing the tree structure of the cross-annotation* `@AmountCheck`.

as shown in the example in Section 2. Figure 11 shows how such an annotated object is tested. The result of the validation is returned as an object of the type `ValidationSummary`, which then can be queried by the application.

Details of the call to the method `validate()` of the `Validator` object, on an object `o`, are shown in Figure 12. The validator goes through all the methods of the object `o`, and by reflection finds all annotations that are marked for validation. Then, for each method with such annotations, all local tests are recursively constructed as a tree, run on the return value and a summary of the outcome is created. The cross-tests are collected and stored in a hash map as we go through the methods, and are run after all methods have been locally tested.

4.1 The validation summary

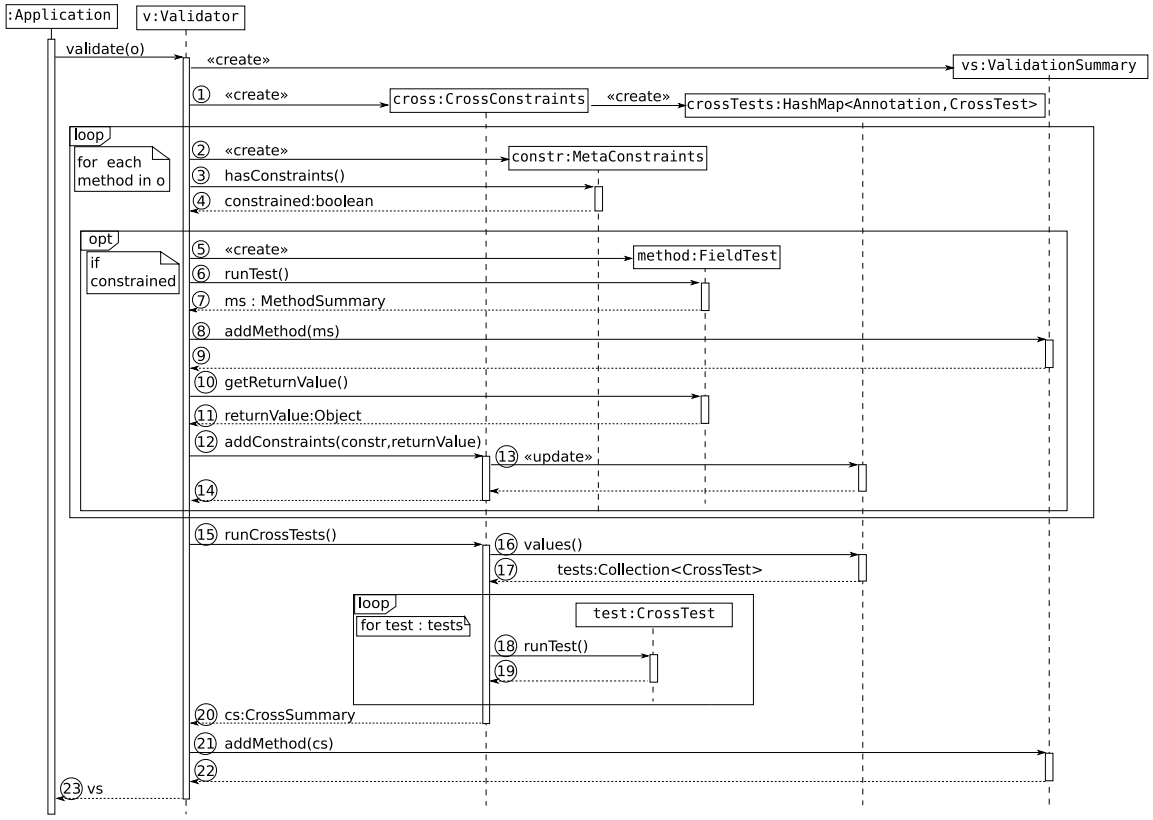
In this section we explain the format of the validation results, as they are returned to the client application. To simplify the explanation, we show in Figure 13 how the cross-annotation `@AmountCheck` defined in Figure 10 is represented in the `ValidationSummary` object.

A `ValidationSummary` object has a tree structure. It contains a vector of `MethodSummary`, one for each method with property-annotations, and one `CrossSummary` object for all the cross-tests. The test in Figure 13 shows the recursive structure of a cross-test in the `CrossSummary`. The boolean composition of the annotation is stored in `SummaryNode` objects, each of which contains a `CrossAnnotation` object storing the annotation itself, and the particular boolean operator associated with its meta-annotations. The class `CrossAnnotation` provides functions to recursively extract cross-annotations, error messages and boolean operators. Each `SummaryNode` mimics the recursive structure of the annotations it represents, and contains a field with the boolean operator used to combine the results of its children. The final result of the composed test is calculated in a bottom-up fashion and stored in the `ValidationSummary` object. The result of the partial test represented by each subtree is also stored in its root. An almost identical procedure is applied to property-annotations in each `MethodSummary`.

In figure 14, we can see how the printed summary is organized in the same logical tree structure of the `ValidationSummary` object. The user can decide the recursive depth of the summary, in order to obtain the desired level of details.

5 Related work

There are alternative ways of tackling the input validation problem, but they are fundamentally different from our approach. For instance, there are static analysis tools [10] which provide support for tainting ([11, 12]) or tools that provide specific solutions for particular input validation vulnerabilities like the AntiSamy project for XSS [13].



1. The `CrossConstraints` keeps track of the cross-tests that should be run on values in the object `o`. The values to be cross-tested are stored in the hash map `crossTests`. For each cross-annotation, a `CrossTest` object contains the list of return values of the methods annotated with it.

Steps 2 to 14 are repeated for each method in the class of the object o.

2-4. `MetaConstraints` is used to extract and store the validation- and cross-annotations on the method. If there are any such constraints, the call to the `hasConstraints()` method returns true. Steps 5 to 14 are only executed if there are constraints.

5-9. The `FieldTest` takes care of retrieving the return value, running the tests on this value, and constructing the corresponding `MethodSummary`. The latter is added to the `ValidationSummary` by the `Validator`.

10-14. The `Validator` retrieves the return value of the method, in order to pass it to the `CrossConstraints`, which, in turn, updates the list of values in the `crossTests` map for each cross-annotation which appears on the method.

15-20. Calling the method `runCrossTests()` causes each `CrossTest` object in the hash map `crossTests` to run the corresponding tests on its list of return values, in a similar way as `FieldTest` does. Finally `CrossConstraints` creates a `CrossSummary`, which extends `MethodSummary`, where each `SummaryNode` contains the result of a cross-test.

21-22. The `CrossSummary` is added as a normal `MethodSummary` to the `ValidationSummary`.

Figure 12: *Sequence diagram for a call to validate on the object* `o`.

```
The value "BICCODE" returned by "getBIC()"
has not passed the following property-test:
+Test: @ValidateBIC() because of:
 |-+Test: @PatMatch(value=\w{8}|\w{11})
=========================================
The following cross-tests have failed:
+Test: @AmountCheck() because of:
 |-+Test: @MaxAmount() because of:
  |-+Test: @OneLessThan(value=1)
  |-+Test: @AllLessThan(value=10000)
+Test: @AllOrNoneNull()
+Test: @ExactlyOneNull() because of:
 |-+Test: @ExactlyNNull(value=1)
=========================================
```

Figure 14: *Printout of the* `ValidationSummary` *for the form in figure 1, given the tests defined in Figures 2, 5 and 9.*

In our framework we replace the classical XML configuration file with annotations. The new implementation provides all the advantages of having an external configuration file, like decoupling of validation logic from the application logic, and reusable tests. In addition, using annotations removes the need for referring to method and class names by string references, which is very error prone and requires additional maintenance. One disadvantage of switching to annotations, might be that runtime changes are not possible anymore. However, being forced to recompile after making changes helps to ensure the type safety of the application.

Additional advantages of using annotations instead of XML configuration files are discussed in Holmgren [8] and Hookom [7]. Both papers also include some technical solutions for using annotations to validate object properties, but only provide some basic illustrative code, rather than a fully functional framework. However, it seems like the ideas in [7] are the starting point for the Hibernate validator and the work in[9].

Our approach has a lot in common with the solutions provided by Hookom and Hibernate. For example, every test is represented by an annotation, and reflection is used to associate tests and methods without explicit string references. Also, a marker-annotation is unavoidable to distinguish an annotation representing a test from other annotations, at least when the user is given the possibility to write custom annotations, as in our framework. On the other hand, if only a fixed set of predefined annotations are made available by the framework, then it is sufficient to check that an annotation belongs to this set.

When it comes to running the actual validation, we are close to the solutions proposed in [7, 9]. In contrast, the solution in Holmgren involves inserting extra code inside the method to be validated. Although this approach allows tests on methods without return values, i.e., setter methods or methods with parameters, it makes the test code and the application code more interdependent, which is what we have tried to avoid.

Composition is also proposed in [9], although the full scope of boolean operators does not seem to have been addressed. What is called a multi-valued constraint in [9], i.e., applying the same annotation with different element values to the same method, can easily be achieved in our framework by encapsulating each instance in another annotation as in Figure 3. Also, support for cross-annotations seems to be lacking.

Struts 2 [3] also provides validation through annotations. It offers a limited set of standard annotations, with no possibility of creating custom tests. As new annotations cannot be created, composition is not possible and the only way to add custom tests is to use a `@CustomValidator` annotation which takes as argument the name of the test. This is then associated to the corresponding class in an XML configuration file. In other words, despite the use of annotations, classes are still referenced to by string names.

The approaches in Struts, Hibernate and Holmgren, all have the disadvantage that, in order to use them in existing projects, substantial changes might be required in the code base. For Struts and Hibernate, the design of the whole project is influenced by the choice of these frameworks. Concerning Holmgren, the code inside

the methods to be tested has to be modified.

Finally, most of these frameworks are mainly designed to work with JavaBeans, and make strong assumptions about the type of applications that will apply them. Our framework, as Heimdall, does not assume much about the application, and should be easy to integrate with any Java project.

6 Conclusion and future works

We have described a framework for using Java annotations to validate input. The source code is available for download [14]. One basic idea in the design of this framework has been that it should be easy to create libraries of custom annotations and tests, and that these tests should be highly reusable. We have tried to offer simple, yet powerful means to do this, like for example, using boolean composition. Besides, we have tried to explore the limits of annotations for input validation by proposing constraints involving interdependent properties, which have not been addressed in any work we are aware of.

However, as we have mentioned earlier, we need to put some limitations on the possible cross-tests that can be created, in favour of composition and reusability. Overcoming these limitations is possible, but the resulting system might become so complicated that using annotations would not be as convenient any more.

For future work, we intend to improve and extend the implementation of the framework with better support for composition and for cross-annotations. In particular, we want to create maps from property-tests to cross-tests, specifying that the property-test should hold for some specified subset of the annotated properties. We will also extend the library of predefined annotations by creating new tests aimed at specific input validation vulnerabilities.

References

[1] "OWASP Top Ten project," May 2009. [Online]. Available: http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

[2] "Commons validator," Apache, May 2009. [Online]. Available: http://commons.apache.org/validator/

[3] "Struts," May 2009. [Online]. Available: http://struts.apache.org

[4] "Hibernate," May 2009. [Online]. Available: https://www.hibernate.org/

[5] L.-H. Netland, Y. Espelid, and K. A. Mughal, "A reflection-based framework for content validation," in *ARES*. IEEE Computer Society, 2007, pp. 697–706.

[6] K. Arnold, J. Gosling, and D. Holmes, *The Java Programming Language, Fourth Edition*. Addison-Wesley, 2006.

[7] J. Hookom, "Validating objects through metadata," O'Reilly, January 2005. [Online]. Available: http://www.onjava.com/lpt/a/5572

[8] A. Holmgren, "Using annotations to add validity constraints to javabeans properties," Sun, March 2005. [Online]. Available: http://java.sun.com/developer/technicalArticles/J2SE/constraints/annotations.html

[9] E. Bernard and S. Peterson, "Jsr 303: Bean validation," Bean Validation Expert Group, March 2009. [Online]. Available: http://jcp.org/aboutJava/communityprocess/pfd/jsr303/index.html

[10] B. Chess and J. West, *Secure programming with static analysis*. Addison-Wesley Professional, 2007.

[11] W. Pugh, "Jsr 305: Annotations for software defect detection," September 2006. [Online]. Available: http://jcp.org/en/jsr/detail?id=305

[12] V. Haldar, D. Chandra, and M. Franz, "Dynamic taint propagation for java," in *ACSAC*. IEEE Computer Society, 2005, pp. 303–311.

[13] "OWASP AntiSamy project," OWASP, May 2009. [Online]. Available: http://www.owasp.org/index.php/Category:OWASP_AntiSamy_Project

[14] BLINDED, "Validator implementation," May 2009.