Dsc+Mock: A Test Case + Mock Class Generator in Support of Coding Against Interfaces

Mainul Islam, Christoph Csallner
Computer Science and Engineering Department
University of Texas at Arlington
Arlington, TX 76019, USA
mainul.islam@mavs.uta.edu,csallner@uta.edu

ABSTRACT

Coding against interfaces is a powerful technique in object-oriented programming. It decouples code and enables independent development. However, code decoupled via interfaces poses additional challenges for testing and dynamic execution, as not all pieces of code that are necessary to execute a piece of code may be available. For example, a client class may be coded against several interfaces. For testing, however, no classes may be available that implement the interfaces. This means that, to support testing, we need to generate mock classes along with test cases. Current test case generators do not fully support this kind of independent development and testing.

In this paper, we describe a novel technique for generating test cases and mock classes for object-oriented programs that are coded against interfaces. We report on our initial experience with an implementation of our technique for Java. Our prototype implementation achieved higher code coverage than related tools that do not generate mock classes, such as Pex.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Symbolic execution, testing tools*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Reliability*

General Terms

Algorithms, Reliability, Verification

Keywords

Mock class generation, method body generation, test case generation, dynamic symbolic execution

1. INTRODUCTION

Testing is a classic dynamic program analysis. It is widely used in software engineering around the world. It consists of finding concrete input values for a piece of code, executing the code on those values, observing the execution, and judging the results. Of these steps, the first one is particularly interesting, as the remaining steps depend on it. I.e., in order to execute a piece of code and to perform any form of dynamic analysis on it, we have to find suitable input values. This requirement applies to code pieces of all levels of granularity. E.g., in order to execute a given statement in a method body, we may need to find the one and only input value of the method that will lead its execution to reach that statement.

To find test inputs, several tools have been developed in recent years, including random test case generators such as JCrasher [2], combinations of static and dynamic analyses such as Check 'n' Crash [3], and dynamic symbolic ("concolic") analysis systems such as Exe, Dart, and Pex [1, 4, 10]. However, existing tools have difficulties with finding good input values for a piece of code that declares that the type of one of its inputs is an interface. This can severely restrict the value of such tools, as interfaces are one of the most powerful abstraction mechanisms in many object-oriented programming languages, including Java and C#. Interfaces support information hiding, separate compilation, subtype polymorphism, and multiple inheritance. When taken together, no test case generator we are aware of supports these important features of object-oriented programming.

In this paper, we present a novel technique for supporting interfaces in test case generation for object-oriented programs. Our technique observes executions of the program under test to collect a detailed set of conditions. The conditions describe the branching conditions encountered, subtype constraints, constraints on the type of input values, and constraints on the values returned from calling an abstract class or interface. Our technique encodes these constraints and issues them to an Smt-solver. From the solution of the constraints, our technique constructs new test cases and program input values. The key difference to previous techniques is that the generated input values may be instances of custom mock-classes that we generate along with the test cases and input values. A generated mock class can implement several interfaces in such a way that the generated method bodies allow our technique to cover pieces of code that cannot be covered with existing techniques.

Specifically, this paper makes the following contributions.

- We provide simple code examples that existing test case generation techniques fail to cover.
- We motivate that such code can be covered with custom generated classes we call mock-classes.

- We present a novel algorithm for generating custom mock-classes.
- We describe an implementation of our technique for Java in our new dynamic symbolic execution engine, Dsc, and report on our initial experimental results.

We describe our examples, design, and implementation in terms of Java programs, but the discussion equally applies to related object-oriented languages, such as C++, C#, etc.

2. MOTIVATION AND EXAMPLES

Coding against interfaces enables us to decouple our code from other code. This has many advantages, including separate compilation and information hiding [9]. I.e., a client may be developed independently of code that the client will be calling at runtime. This is accomplished by coding the client against interfaces, thereby representing abstractly the classes that will be called at runtime. In this way, the client may be developed before such service classes are completed. To support such independent development, in addition to compiling code independently, we also want to test code independently. To test the client class, we need to find a set of classes that implement the interfaces referred to by the client. Oftentimes, such classes do not (yet) exist. This means we need some test case generator to not only generate test cases but also classes that implement any interfaces used by the client. This task is complicated by the fact that in several cases we need classes that implement more than one interface. This is a common requirement. Classes in modern object-oriented languages often implement multiple interfaces, including many classes of the standard libraries of Java and C#.

```
public class C {             // client
  public void foo(I i, int x) {
    int y = i.m1(x);
    if (i instanceof J) {
      J j = (J) i;
      int z = j.m2(x);
    }
    //..
  }
  //..
}

public interface I {
  public int m1(int x);
}

public interface J {
  public int m2(int x);
}
```

Listing 1: C is a client of two interfaces, I and J. To test the client, we need an implementation of both interfaces.

For illustration, we use the simple example given in Listing 1, in which class C is a client of two interfaces, I and J. Specifically, C's foo method has a parameter i, whose declared type is the I interface. The foo method uses i, by first calling the m1 method on it and later possibly the m2 method. Those two methods, m1 and m2, are declared by different interfaces, I and J. In order to call both methods, we need to find an instance of I that is also an instance of J. If there are no implementations (yet) of I or J, a test case generator has to provide them. However, existing test case generators we are aware of, including JCrasher, Check 'n' Crash, and Pex [2, 3, 10], cannot do that. Instead, they only generate an implementation of the I interface or find the null reference.

2.1 Implementing the Right Interfaces

From the above example of Listing 1, it may seem that generating an implementation of multiple interfaces is straightforward: Why not generate a class that implements all interfaces referred to by the code? However, this will generally not work. Like other boolean-valued expressions, a subtype constraint may appear nested in the path condition, for example, within a negation. Listing 2 gives an example, in which class K implements all interfaces referred to by the client class C. However, plainly implementing these interfaces is not sufficient for covering method bar of the client.

```
public class C {  // client, continued
  //..
  public void bar(I i) {
    if (i==null)
      return;
    if (! (i instanceof J))
      //..
  }
}

public class K implements I,J {
  //..
}
```

Listing 2: Class K implements all interfaces referred to by client class C, but fails to cover the bar method.

2.2 Generating Meaningful Method Bodies

For the two previous examples it is sufficient to generate a mock-class that implements the right interfaces. But there are other cases for which this strategy will not suffice. To reach a given piece of code, we may have to generate a method body that contains a certain sequence of statements. Listing 3 gives one such example. In order to cover the entire foobar method, we have to generate a particular implementation of the m1 method.

```
public class C {  // client, continued
  //..
  public void foobar(I i) {
    int b=5;
    if (i.m1(b) == b+1)
      //..
  }
}
```

Listing 3: Only certain implementations of the I.m1 method can cover the entire foobar method.

3. BACKGROUND

In this section we provide necessary background for our mock-class generation technique, specifically, on Java reference types, their subtype relation, dynamic symbolic execution, and the dynamic symbolic execution framework in which we implement our solution.

3.1 Sub-/Supertype Relation in Java

In Java every variable and every expression has a type that can be determined at compile time [5]. Primitive types such as int and boolean are distinguished from reference types. Reference types include interfaces and classes. In this paper we are mainly interested in reference types.

Like many languages, Java defines a binary subtype relation on reference types. For example, if a Dog type declares that it implements or extends an Animal type, then Animal is a direct supertype of Dog and Dog is a direct subtype of Animal. Reflexive and transitive closure of these "direct" relations yield the super- and subtype relations.

A class has one direct class supertype and arbitrarily many interface supertypes. The Object class is a prominent exception, it has no direct supertype. Another special type is the null type; it has no direct subtype but is a subtype of every other type. An interface only has interface supertypes, except if it does not declare any supertypes, then it has an implicit one—the Object class.

3.2 Dsc Dynamic Symbolic Execution Engine

Symbolic execution [6] is a systematic path exploration where numeric constraints are generated. Specifically, symbolic execution replaces the concrete inputs of a program unit (typically, a method) with symbolic values, and simulates the execution of the program so that all variables hold symbolic expressions over the input symbols, instead of values.

Dynamic symbolic execution [4, 1] is a variation of conventional symbolic execution which starts exploring paths while executing the program in test. Usually, to generate test cases, symbolic execution is combined with a constraint solver to solve the constraints discovered during path exploration. The constraints are gathered, in parallel to symbolic execution, from predicates in branch statements of the code that is analyzed. A constraint solver is used then to compute different variations of the inputs which directs future program executions along all feasible paths.

Dsc is a dynamic symbolic execution engine for Java, in which we have implemented our approach for generating mock-classes. Dsc uses the high-performance automated theorem prover Z3 from Microsoft Research [8], to solve constraints generated during symbolic execution. It is not the only tool for dynamic symbolic execution. But Dsc is the first and only one tool we are aware of that generates mock-classes for dynamic symbolic execution.

4. SOLUTION DESIGN

We begin the description of our solution with a high-level overview. Like in dynamic symbolic execution, our approach consists of a loop. In each iteration, we try to cover a new execution path. Following are the steps taken in each iteration.

1. Execute the program on a given (or random) input on a dynamic symbolic engine: Collect control-flow constraints and invert one of them to obtain the path condition for the next execution.
2. Map each encountered reference type to a non-negative integer value.
3. Issue the new path condition to the constraint solver.
4. Encode the subtype relation of the encountered reference types and issue it to the constraint solver.
5. If path condition plus subtype constraints are not satisfiable, revoke subtype relation from constraint solver.
6. Introduce mock-classes, add them to our encoding of the subtype relation, and issue the new subtype relation to the constraint solver.
7. If the constraint is satisfiable, determine the super types of each mock class and generate the new mock-classes.

4.1 Subtype Relation Matrix

We encode the subtype relation of the types we discovered along an execution path in the form of a matrix. Table 1 shows an example, the full (reflexive and transitive) subtype relation for the types of Listing 1 plus mock class M that we introduce to generate a test case that will cover the entire method. MC, MI, and MJ are boolean-valued variables. Their values will be fixed by the constraint solver as part of the solution of our constraint system.

		Object	C	M	I	J	null
1	Object	x					
2	C	x	x				
3	M	x	MC	x	MI	MJ	
4	I	x			x		
5	J	x				x	
0	null	x	x	x	x	x	x

Table 1: Example reference type encoding and subtype relation matrix. The first column shows our encoding of reference types in non-negative integer values. In the remaining columns, an x at position [A,B] means that type A is a subtype of the type B, an empty field that A is not a subtype of B. For example, the x entries in the Object column represent that each reference type is a subtype of java.lang.Object. MC, MI, and MJ are boolean variables.

4.2 Building Constraints and Generating Mock-Classes

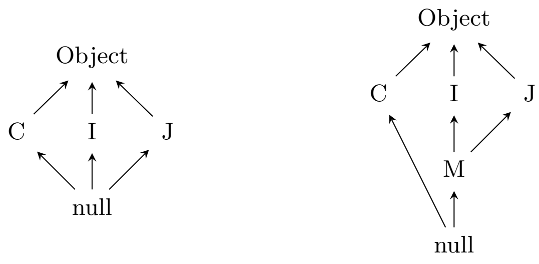

Figure 1: Left: Direct subtype relation of the original program of Listing 1. Right: A desired solution.

Figure 1 shows the direct subtype relation as a graph. For Listing 1, following are the constraints we want to solve:

- type(i) subtypeof I
- type(i) != null type
- type(i) subtypeof J

First, we send the constraints to the constraint solver to check if the constraints can be satisfied with the existing types. But no existing type may satisfy this constraint system. If this is the case, we introduce one new type—a mock-class—for each reference variable. For the example of Listing 1, we introduce mock-class M. Each mock-class gets its unique class id.

The dynamic type of the reference variable can now range over all classes, including the mock-class. Depending on the type-constraints, the mock-class can assume any position in the subtype-lattice—it can have any combination of super types that is legal according to the type rules of the programming language. The constraint solver can decide where in the lattice to place the mock-class. For example, M may implement an arbitrary number of interfaces.

We model the relevant type rules of the Java language in our constraint system. For example, in Java, a class can only have one direct super-class. To model this type rule, we represent the super type of the mock class with a variable and restrict its range to the class types.

We issue the new, larger constraint system to the constraint solver. In this example, Z3 returns a solution that sets both MI and MJ to true. So, we finally generate a concrete mock-class M that implements interfaces I and J.

4.3 Generating Mock-Class Method Bodies

A mock-class typically implements an interface, which is a group of related abstract methods (methods with empty bodies). Determining the possible body of a method, especially the return value of the method, is an important part of mock-class generation. Here, we are specifically interested in methods that have a non-void return type. The value returned by a method often plays a role in subsequent branching decisions of the caller. So, while generating a method body, we focus on finding the possible return values of the method.

1. Find a list of methods that a mock-class needs to implement from the solution described in Section 4.
2. Generate empty method bodies with a default return value (e.g., a 0 if return type is integer) for each of the methods in the list when a new mock-class is generated.
3. In each invocation of a mock-class method introduce a symbolic variable for the method return value.
4. Build new constraints with the new variable and issue the constraints to the constraint solver.
5. The constraint solver will determine a possible value for the variable in most common cases. Retrieve the value from the solution.
6. In the next iteration, return the value given by the constraint solver from the mock-method.

According to the above algorithm, an instance of a mock-method returns the same value in all cases. The return value is determined by the constraint solver. Therefore, this approach works for simple cases in which it is adequate to satisfy a path condition with a single value. The program presented in Listing 3 is an example of such a case. In this example, it is enough to return a value from method "m" that is equal to "b+1" in order to reach the goal. But there are cases in which a method needs to return different values in different cases to satisfy a path condition. We are extending our approach to handle more complex cases and describe it briefly in Section 7.

5. IMPLEMENTATION

In this section, we describe the implementation of our algorithm to generate a mock-class. First we briefly describe how we have implemented the subtype relation matrix and determined the type of a mock-class. Then we describe the generation of a mock-class method.

5.1 Type Encoding

We number Java types in the order we discover them. Each time we dynamically discover a new class or interface during dynamic symbolic execution of a program path, we assign a new non-negative integer value to that type. The discovery may happen due to one of many events such as symbolically executing a static field access.

We number classes and interfaces separately and pre-fill our type encoding with two special Java types. I.e., we assign 0 to the null type and 1 to the Object class.

5.2 Determining the Mock-Class Type

We encode the existing types by numbering them from 0,1,. . . ,N. As we have described in Section 5.1, the first two numbers 0 and 1 represent the null type and the Object class respectively. We use a two dimensional array "SuperTypes[0..N][0..N]" to represent the subtype relation matrix. Here N is the number of reference types (classes and interfaces) in an execution path at any moment. The size of the array can be extended while new reference variables are introduced. We initialize the subtype relation matrix with the followings:

- $SuperTypes[0][0..N] \leftarrow true$, to represent that the null type is a subtype of every other type.
- $SuperTypes[0..N][1] \leftarrow true$, to represent that every type is a subtype of the Object class.
- $SuperTypes[M][M] \leftarrow true$, where $0 \leq M \leq N$, to represent that each type is a subtype of itself.

Then we model properties of the sub-class and sub-type relations of a mock-class as follows:

1. When a mock-class "M1" is introduced, it must have exactly one direct super class. To implement this we generate a Z3 variable: "M1_Var". Then we apply a constraint: $1 \leq M1\_Var \leq N$. The constraint solver determines the value of M1_Var.
2. To implement transitive closure, we copy the "true" values from the direct super class of "M1" in the subtype relation matrix.
   $SuperTypes[M1][1..N] = SuperTypes[M1\_Var][1..N]$.
3. Reflexive closure is implemented with:
   $SuperTypes[M1][M1] \leftarrow true$.
4. "M1" may implement an arbitrary number of interfaces. To implement this, if there are I1,I2,..,In interfaces, we have left $SuperTypes[M1][I1..In]$ unconstrained, where $1 \leq I1..In \leq N$. The constraint solver determines which of these interfaces need to be implemented by "M1".

When the subtype relation matrix is satisfiable by the constraint solver, we retrieve the super types from the solution and generate the new mock-class accordingly.

5.3 Generating Mock-Class Methods

When we know the interfaces a mock-class needs to implement, for each method in each interface we generate an empty method body. In each method body, we return a default return value. I.e., we return a 0 (zero) if the return type of a method is integer. So far, we have considered only methods that return an integer. Handling other return types is left for future work.

In each invocation of a mock-class method we introduce a symbolic variable. To determine if the current method is a mock-class method we simply check if the current class is a mock-class. We can do that easily, because while generating mock-classes we add a unique prefix to the class name. We create a map to keep track of the new variables generated for the different mock-methods. We also generate a unique name for each new variable. The new symbolic variable is then used to build the constraints. The constraint solver determines a value for each new variable. In the next invocation of a mock-method we determine the value of the symbolic variable that was introduced for the current method in the previous iteration. We can do that easily from the map we created and with the unique names of the variables.

6. PRELIMINARY RESULTS

In this section, we report our initial experience in using our prototype test-case and mock-class generator. We first compare our test-case generator, Dsc, when running with and without mock-class generation. We also compare Dsc with its leading competitor, Pex [10]. We ran Dsc on the motivating examples of Section 2 and on some real-world examples that use an open source API called Java Messaging Service, JMS. JMS allows Java applications to create, send, receive, and read messages. We took all measurements on a 2.26GHz Core2 Duo processor machine.

6.1 Motivating Examples of Section 2

For our motivating example of Listing 1, without mock-class generation, our dynamic symbolic execution engine cannot reach any of the two method calls (0/2). Dsc just generates a default input value of null, which does not reach any of the calls. When enabling our mock-class generator, our dynamic symbolic execution engine reaches all goals (2/2). For our second example, we obtain similar results. With mock-class generation enabled, we can cover the entire method. Without mock-classes, we cannot. With or without mock-classes, the exploration of both examples took a similar amount of time of about half a second. In summary, with mock-classes we achieve full coverage of the simple example programs. None of the existing dynamic symbolic execution tools we are aware of can cover these cases.

6.2 JMS Examples

JMS provides a list of interfaces that makes it easy to write business applications those asynchronously send and receive business data and events. We have used the Java Message Service (Version 1.0.2b) interfaces to write our examples. As a preliminary experiment we have written 6 methods which take different JMS interfaces as arguments. Our focus in writing these methods was to define as many goals as possible which can be reached only by using mock-classes. On an average we have defined 5 goals per method. Then we have applied dynamic symbolic execution with and without mock classes to each of these methods. Table 2 lists the number of goals reached for each of the methods with and without mock-class generation.

Interfaces used from JMS	#goals	#reached w/o mock classes	#reached with mock classes
BytesMessage	5	0	5
ConnectionMetaData	5	0	5
MapMessage	4	0	4
Message	6	0	6
MessageProducer	4	0	4
StreamMessage	4	0	4

Table 2: Each row of the table shows the results for one of the methods we have tested. The first column contains the JMS interfaces used in each method.

Following is one example test method called "ByteMessageTest". It takes a JMS interface "BytesMessage" as a parameter. We have defined 5 goals in the method. When running without mock class generation, none of these goals is reached. When running with mock class generation, our generated test cases reach all five.

```
public void ByteMessageTest(BytesMessage bm) {
  int len = 10;
  if(bm.readUnsignedByte() == len+1)
    { /* goal 1 */ }
  if(bm.readByte() == len+1)
    { /* goal 2 */ }
  if(bm.readShort() == len+1)
    { /* goal 3 */ }

  byte[] b = new byte[len+1];
  if(bm.readBytes(b) == len+1)
    { /* goal 4 */ }
  if(bm.readBytes(b, b.length) == len+1)
    { /* goal 5 */ }
}
```

6.3 Comparison with Pex

We have performed an initial comparison with the most advanced dynamic symbolic execution engine we are aware of, Pex. Dsc and Pex are similar in that they both use dynamic symbolic execution to generate test cases for object-oriented programs. While Dsc targets Java, Pex targets .Net programs. To compare Dsc with Pex, we translated our motivating examples of Section 2 to C# and ran them on the latest version of Pex (version 0.91.50418.0).

For the motivating example of Listing 1, Pex generates the same input value (the default value null) as Dsc without mock-class generation. Pex by itself cannot reach any of the two method calls (0/2). When it is used with their recent extension, Moles [7], it can reach the first method call (1/2). In comparison, Dsc with mock-class generation reaches both calls (2/2). For the second example, we got similar results. Pex generates input that cannot cover the entire program, just as Dsc without mock-class generation. In summary, with mock-class generation enabled, for the two simple motivating examples, Dsc achieves higher code coverage than Pex.

7. ONGOING WORK

Currently we are extending our algorithm, to handle more complex cases, such as the example given in Listing 4.

```
public class C {  // client, continued
  //..
  public void foobar(I i, int p) {
    int b=5;
    if (i.m1(p) > b && i.m1(p+10) < b)
      // goal 5
  }
}
```

Listing 4: Our current implementation cannot reach goal 5

For the example in Listing 4, we cannot reach "goal 5" if "m1" always returns the same value. In other words, we cannot satisfy both the conditions "i.m1(p) > b" and "i.m1(p+10) < b" with a single return value of "m1". When same inputs are applied, it is expected that a method will always return the same value. But for different inputs, it is common that a method might return different values. Both conditions in the above program can be satisfied if we return two different values from method "m1" for two different input parameters. This is the basis of our current work to generate mock-class methods for such more complex cases. The main part of this algorithm is as below:

1. In each invocation of a mock-class method, introduce one symbolic variable each for the return value and each method parameter.
2. Build new constraints with the newly introduced variables and issue the constraints to the constraint solver.
3. The constraint solver will determine possible values for the variables.
4. In the next iteration, generate a method body using different combinations of the values produced by the constraint solver.

If we apply this algorithm to the above example, two new variables will be introduced for each invocation of method "m1". Also for the parameters of the method, two more variables will be introduced. If these variables are called "returnVar_1", "returnVar_2", "param_1" and "param_2" respectively, then we will get the following constraints:

- p = any arbitrary integer
- b = 5
- returnVar_1 > b
- returnVar_2 < b
- param_1 = p
- param_2 = p+10

For the above constraints, the constraint solver will possibly come up with a solution such as the following:

- p = 0
- b = 5
- returnVar_1 = 6
- returnVar_2 = 4
- param_1 = 0
- param_2 = 10

We can now combine these values to generate a body of method "m1" as shown in Listing 5 and use it in the next iteration of the symbolic execution.

```
public class MockC implements I {
  public int m1(int x) {
    if (x == 0)
      return 6;
    else if (x == 10)
      return 4;
    else
      return 0;
  }
}
```

Listing 5: A possible method body for "m1" to reach goal 5 of the example in Listing 4.

8. CONCLUSIONS

We described a novel technique for generating test cases and mock classes for object-oriented programs that are coded against interfaces. We reported our initial experience with an implementation of our technique for Java. Our prototype implementation achieved higher code coverage than related tools that do not generate mock classes, such as Pex.

The source code for our experiments is available at `http://cseweb.uta.edu/~mainul/mockclasses/`

9. REFERENCES

[1] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: Automatically generating inputs of death. In *Proc. 13th ACM Conference on Computer and Communications Security (CCS)*, pages 322–335. ACM, Oct. 2006.

[2] C. Csallner and Y. Smaragdakis. JCrasher: An automatic robustness tester for Java. *Software—Practice & Experience*, 34(11):1025–1050, Sept. 2004.

[3] C. Csallner and Y. Smaragdakis. Check 'n' Crash: Combining static checking and testing. In *Proc. 27th ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 422–431. ACM, May 2005.

[4] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223. ACM, June 2005.

[5] J. Gosling, B. Joy, G. L. Steele, and G. Bracha. *The Java Language Specification.* Prentice Hall, third edition, June 2005.

[6] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[7] Microsoft Corporation. *Unit Testing with Microsoft Moles*, Mar. 2010. http://research.microsoft.com/en-us/projects/pex/molestutorial.pdf.

[8] L. d. Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340. Springer, Apr. 2008.

[9] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, Dec. 1972.

[10] N. Tillmann and J. de Halleux. Pex - white box test generation for .Net. In *Proc. 2nd International Conference on Tests And Proofs (TAP)*, pages 134–153. Springer, Apr. 2008.