Testing Object-Oriented Programs using Dynamic Aspects and Non-Determinism

Michael Achenbach
University of Aarhus, Denmark
ma@cs.au.dk

Klaus Ostermann
University of Marburg, Germany
kos@informatik.uni-marburg.de

ABSTRACT

The implementation of unit tests with mock objects and stubs often involves substantial manual work. Stubbed methods return simple default values, therefore variations of these values require separate test cases. The integration of mock objects often requires more infrastructure code and design decisions exposing private data. We present an approach that both improves the expressiveness of test cases using non-deterministic choice and reduces design modifications using dynamic aspect-oriented programming techniques.

Non-deterministic choice facilitates local definitions of multiple executions without parameterization or generation of tests. It also eases modelling naturally non-deterministic program features like IO or multi-threading in integration tests. Dynamic AOP facilitates powerful design adaptations without exposing test features, keeping the scope of these adaptations local to each test. We also combine non-determinism and dynamic aspects in a new approach to testing multi-threaded programs using co-routines.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques

Keywords

Testing, mock objects, non-determinism, dynamic aspect-oriented programming, Ruby

1. INTRODUCTION

In Test-Driven Development, tests represent design requirements specified before implementing the system under test [4]. Unit tests focus on analysis and verification of small, incomplete subsets of the overall system. In an object-oriented context, the classes under test often collaborate with very complex classes that are replaced and imitated by mock objects or stubs, which cover only a small part of the original classes' functionality and observe their collaboration [5, 15].

The test process is influenced by several factors: (1) the precision by which a mock object observes class collaborations and imitates its corresponding original class and (2) the engineering amount and design decisions for integrating such objects. In this work, we suggest improvements for both issues: (1) by introducing non-determinism into mock and stub configurations to gain more expressive test cases and (2) by using dynamic aspect-oriented programming (dynamic AOP) techniques for a seamless and flexible integration of mocks and stubs.

A traditional test case with collaborating mock objects and objects under test corresponds to exactly one execution. Each new execution with, e.g., different stubbed return values or different collaboration configurations needs separate test artifacts, like separate test methods with different initializations of mocks. We suggest in this work to broaden this approach by performing several related executions within one test case. The separation of executions is defined by non-deterministic choices in stubbed methods or as test input. E.g., instead of returning simple defaults like the value `false`, choice sets like `{true,false}` can be specified. The tests can then be executed in an execution environment that provides non-deterministic choice like an explicit state model checker or a similar program exploration tool [18].

Our second improvement focuses on the engineering extent and design modifications required when instantiating and injecting mock objects into the system under test. In general, the system design should include all testing concerns [6], but we argue that this often leads to interfaces exposing private data or requiring additional parameterization that only addresses test features. Furthermore, including test objectives into the design might lead to a higher engineering extent in an object-oriented setting.

To reduce this engineering extent and the size of the code base required for testing, we suggest dynamic AOP techniques in this work. While static[1] AOP techniques have been suggested before to ease the integration of mock objects [13, 9], we introduce dynamic AOP with powerful dynamic scoping to increase test flexibility with mock objects while keeping design adaptations local to the test.

[1]By static AOP we mean AOP approaches where pointcuts are specified in terms of static syntax. In dynamic AOP, static AOP is augmented with the possibility to incorporate dynamic information, either via dynamic pointcuts or dynamic deployment.

ECOOP '10 Maribor, Slovenia

The contributions of this work are as follows:

- We present ExploR, a new framework for program exploration using non-deterministic choice in Ruby. We demonstrate by example our approach of increasing expressiveness of unit testing with mock objects using the non-deterministic choice operator.
- We introduce a new approach using dynamic AOP for test local design adaptations and injection of mock objects. The approach is illustrated using TwisteR, a library for dynamic deployment of aspects supporting powerful scoping techniques that keep the influence of aspects local to each test [3].
- We combine both techniques in a new approach to testing multi-threaded programs. We use dynamic AOP to transform a non-deterministic program into a deterministic one in order to perform reproducible tests. We then reintroduce a controlled non-determinism using ExploR for a more expressive under-approximation of the real behavior.

The remainder of this paper is organized as follows: We introduce non-determinism and dynamic AOP for testing in Sections 2 and 3 respectively. Section 4 illustrates both approaches for testing multi-threaded programs followed by a brief implementation overview in Section 5. We discuss related work in Section 6 followed by our conclusion.

2. TESTING WITH NON-DETERMINISM

Traditionally, one test case performs exactly one execution. In this work, we broaden this approach by grouping several related executions within one test case, focusing on the configuration of mock objects and method stubs. This relation is twofold: Using independent grouping, the anticipated outcome is not expected to differ among several executions, while the verification of this expectation is a test objective. Using dependent grouping, different input output pairs are enumerated.

With classical tools like jMock, a mock object can be generated according to an interface [6]. Mock objects combine the concept of *expectations*, meaning the observation of command method calls and the verification of these observations against a predetermined specification, and the concept of *stubs*, meaning predefined return values or simple defaults for stubbed query methods. Expectations, like an expected sequence of method calls, are defined on the mock object beforehand and implicitly verified after running the test. Stubbed methods often return simple defaults, like `false` for a boolean value. They can also yield a particular value or a sequence of values in subsequent calls.

We use the concept of non-deterministic choice to model a fork in the execution flow. Such a choice is represented by a concrete language construct, as we will show in the following examples. The examples are written in the Ruby programming language, but the general concept can be applied to other languages as well. The non-deterministic choice operator is implemented as a library method of our program exploration framework ExploR. We implement mock objects using the Mocha library for Ruby [1].

Figure 1 shows two classes in a sales scenario. The system under test is the `SalesManager` class, that in a simplified version only calculates the price for a list of products.

```
class SalesManager
  def initialize ps; @ps = ps; end

  def calc_price products, ps = nil
(1)   ps ||= @ps
      sum = 0
      products.each{ |p|
(2)     sum += (p.price * p.number * ps.discount(p))
      }
      sum * ps.clearance_discount
  end
end

class PricingStrategy
  def discount product; end
  def clearance_discount; end
end
```

Figure 1: Sales example

The constructor (`initialize`) can be parameterized with a pricing strategy, stored in field `@ps`. Method `calc_price` is parameterized with a list of products and an optional pricing strategy. If no strategy is given, the one stored in `@ps` is used at (1). Each product has a price and a quantity, used to calculate the whole price at (2) using different discount factors.[2] E.g., a factor can be 1.0 for no discount or 0.8 for a 20% discount, etc. The methods of the pricing strategy are also illustrated in Figure 1 with a pseudo-interface.

2.1 Grouping Independent Tests

Our first type of execution grouping is shown in Figure 2. The test objective is to test method `calc_price` with different configurations of mocked pricing strategies. At (2), a list of products is initialized, where each product is created using the `stub` method of the Mocha library. It returns a fresh object that responds to the methods `price` and `number` with the specified return values. At (1), a mock object for the pricing strategy is created with fixed values for both discount types. The expectation specification is part of the Mocha library. Method `discount`, e.g., is expected to be called zero or more times with any parameter from the list of specified products. When instantiating the class under test with the mock object at (3), the return values of `discount` and `clearance_discount` are defined using the non-deterministic choice operator `choice`, that selects between its arguments 1.0 and 0.5, meaning no discount or a 50% discount respectively. Finally, the method under test is called and the outcome is tested against a loose specification at (4), involving an approximation of the price without discount.[3] The expectations of the mock objects are verified implicitly at the end of the test. The specified non-deterministic choices lead to four separate executions of the test code.

2.2 Grouping Dependent Tests

In the second type of execution grouping, the non-deterministic selections depend on each other. In Figure 3, the test from Figure 2 is modified with a dependent selection

[2]The `each` function in Ruby iterates over an array and executes in each turn the subsequent block of code between the curly braces, parameterizing the variable `p` with the current product.

[3]The `inject` method of Ruby is similar to a *fold*.

```
    def mock_pricing_strategy products, disc, clearance
(1)   pricing_strategy = Object.new
      pricing_strategy.expects(:discount)
        .at_least(0).with(any_of(*products))
        .returns(disc)
      pricing_strategy.expects(:clearance_discount)
        .returns(clearance)
      return pricing_strategy
    end

(2) prods = [stub(:price=>10, :number=>2),
      stub(:price=>2, :number=>6)]

(3) s = SalesManager.new mock_pricing_strategy(
      prods,
      choice(1.0, 0.5),
      choice(1.0, 0.5))

    price = s.calc_price(prods)
    approx = products.inject(0){|r, x| r + x.price*x.number}

(4) assert 0 <= price
    assert price <= approx
```

Figure 2: Testing the sales example with independent choices

```
(1) p1_price, expected_price = choice [10, 32], [5, 22]
    prods = [stub(:price=>p1_price, :number=>2),
      stub(:price=>2, :number=>6)]

    s = SalesManager.new mock_pricing_strategy(prods, 1.0, 1.0)
    price = s.calc_price(prods)

(2) assert price == expected_price
```

Figure 3: Testing the sales example with dependent choices

at (1).[4] The first product is now stubbed with the non-deterministic `p1_price`, while the mocked pricing strategy has a constant configuration. At (2), the outcome is verified against `expected_price`, that is non-deterministically instantiated with a value corresponding to each `p1_price` choice.

Our approach is comparable to parameterized test cases, yet our test cases are more compact, since the parameterization is done in place. Parameterized test cases also need an external driver that iterates over different test inputs, while such a driver is implicit in the non-deterministic choice operator.

2.3 Other Applications of Non-Determinism

The previous sections introduced non-determinism for deterministic applications to subsume a larger number of executions without code duplication in unit tests. Non-determinism can also be used to create complex test input like object graphs [10] or to model naturally non-deterministic parts of a program, like the exception mechanism of IO features [14]. In integration testing, it is often necessary to decouple and simulate non-deterministic features like IO, system calls and multi-threading.

[4]Comma-separated variables are parallelly assigned in Ruby. The choice method on the right-hand side chooses between one of the two arrays and evaluates to an array of two values, which are projected to the two variables.

3. TESTING WITH DYNAMIC AOP

Several approaches call for the use of aspect-oriented programming (AOP) techniques[5] in connection with mock objects [13, 9]. While Freeman et al. recommend interfaces that include both production and test features for mock injection [6], we argue that this approach might lead to the exposure of local and private information. Furthermore, some dynamic scripting languages like Ruby do not provide a Java-like interface concept and programmers often implement class collaborations in an ad-hoc manner.

In earlier work, we analyzed the applicability of AspectJ [11] for integrating abstractions of program entities into a test environment [2]. While the approach is usable for mock integrations that have a global scope, it turned out to be insufficient for, e.g., usage of different mock objects in different dynamic contexts. Furthermore, in testing, each test method has its own dynamic scope that requires different aspects. With static AOP like AspectJ, the separation of many different dynamic contexts leads to numerous control flow checks, scattered over one aspect [17]. We therefore propose the application of dynamic AOP and powerful scoping mechanisms with per object deployment of aspects using the dynamic aspect language TwisteR [3].

3.1 Test-Local Dynamic Adaptations

Figure 4 extends the sales example from Figure 1 by subclassing the `SalesManager` class. The method `best_price` tries to find an optimal pricing strategy among several clearance and product discount configurations. The pricing strategy is instantiated locally. The `ComparisonSalesManager` calculates the minimum price of different sales managers, while the `BestClearance` and the `BestProduct` respectively calculate the best price according to a general clearance discount on the whole sale or different discount models for each product group. The pricing strategies are instantiated based on local data, but in more complex examples they might depend on other external services. Our following test objective is to analyze the collaboration of the several sales manager classes and `PricingStrategy` and to verify some upper and lower bounds of the price comparison. Firstly, we want to mock the pricing strategy with different mock configurations for different sales managers without exposing a parameter in `best_price`. Secondly, any design modification by aspects should not exceed the dynamic scope of one test.

Figure 5 illustrates our test method. We reuse the method `mock_pricing_strategy` from Figure 2 to initialize the mock object. Dependent on the dynamic context of the initialization of `PricingStrategy`, we will use a different mock configuration. This is captured in the two aspects at (2) and (3). Each of the aspects intercepts[6] the creation of `PricingStrategy` and instantiates different mock objects for clearance and product discount calculations, dependent on

[5]AOP separates concerns of programs by applying *pointcut* and *advice*. Possible interference points (*join points*) are specified for the program text (static) or for the execution (dynamic). *Pointcuts* are program constructs that select join points, *advice* is code, executed at particular join points. *Aspects* modularize and associate pointcut and advice.

[6]Aspects in TwisteR are first-class values, their definition is passed as a block of code (between `do..end`) to the constructor (called by `Aspect.new`). The `around` keyword specifies a so called around advice that will be executed at join points selected by the pointcuts `new & name(...)` — object creations of class `PricingStrategy`.

```
class ComparisonSalesManager < SalesManager
  def initialize s1, s2; @sales1, @sales2 = s1, s2; end
  def best_price prod
    [@sales1.best_price(prod), @sales2.best_price(prod)].min
  end
end

class BestClearance < SalesManager
  def best_price prod
    calc_price prod, PricingStrategy.new(0, 1.0, 0.8)
  end
end

class BestProduct < SalesManager
  def best_price prod
    [[3, 0.7], [5, 0.6]].map{ |n, discount|
      calc_price prod, PricingStrategy.new(n, discount, 1.0)
    }.min
  end
end
```

Figure 4: Extended sales example

some of the original constructor arguments.[7] Both aspects are deployed on the different sales manager objects using the pervasive scoping strategy initialized at (1). Pervasive scoping causes the aspects to interact with every further method call to `bc` and `bp`, and each further object created and method called in that dynamic extent.[8] The aspect will not affect anything outside this dynamic extent, e.g., neither the `assert` methods at the end of the test, nor other test methods.

Running the test, the first mock configuration will be used to instantiate a mock object instead of `PricingStrategy` in the `best_price` method of the `BestClearance` class, while a different mock configuration will be used in the context of the `BestProduct` class. Like before, the expectations of all instantiated mock objects will be implicitly verified at the end of the test. Due to this design, the local pricing strategies do not need to be exposed as parameters of the `best_price` methods, since the specified aspects perform the mock integration locally. Dynamic AOP facilitates the usage of different configurations within the same test, also providing explicit control over the borderlines of aspect influence. There is no need for clean-up before or after each test, since all design adaptations are local.

3.2 Other Applications of Dynamic AOP

Another scenario for dynamic deployment is the application of *delayed choice*, where a non-deterministic choice is not evaluated directly, but at a later execution point, when its value is used [7]. E.g., the choice in Figure 2 at (3) is evaluated when calling `mock_pricing_strategy` and not when the mocked method `discount` is called during the test. Such a preliminary non-deterministic decision leads to a combinatorial duplication, even if the `discount` method is not called at all. To delay the choice, it can be boxed within a closure: `lambda{choice(...)}`. We then need to instrument the code returning such delayed choices with a dynamic aspect that applies the closure, e.g., at method returns. Further

[7]The `args` keyword within the advice gives access to the original arguments of the object creation, e.g., `args[0]` is the first argument of the constructor call of `PricingStrategy`.

[8]For more information on dynamic scoping we refer the reader to [3, 17].

```
products = [stub(:price=>10, :number=>2),
  stub(:price=>2, :number=>6)]
bc = BestClearance.new nil
bp = BestProduct.new nil
manager = ComparisonSalesManager.new bc, bp
(1) pervasive = Strategy.new {[True,True,True]}

(2) clearance_aspect = Aspect.new do
    around new & name('PricingStrategy') do
      mock_pricing_strategy products, 1.0, args[2]
    end
  end
  deploy_on bc, clearance_aspect, pervasive

(3) product_aspect = Aspect.new do
    around new & name('PricingStrategy') do
      mock_pricing_strategy products, args[1], 1.0
    end
  end
  deploy_on bp, product_aspect, pervasive

  price = manager.best_price(products)
  assert price > 0
  assert price < 32
```

Figure 5: Test harness with per object deployment of aspects

applications of dynamic aspects are replacement of features like multi-threading, injection of non-deterministic choice to choose thread interleavings, and class replacements in different contexts in general [2].

4. MULTI-THREADED PROGRAMS

Multi-threaded programs impose a natural non-determinism into the inner workings of a program, while the overall outcome is intended to be deterministic. Testing multi-threaded programs is a non-trivial task, since non-determinism has to be modeled or replaced in a deterministic fashion, often requiring specialized interpreters. We propose an approach to testing multi-threaded programs, where threads are replaced by co-routines using context switches with continuations.[9] We additionally model a subset of the scheduling choices of the thread scheduler using non-deterministic choices in our framework. We combine both approaches of Section 2 and 3, dynamic AOP for the replacement of the original thread features and non-determinism for an under-approximation of thread interleavings.

Figure 6 shows a simple code example that can produce a deadlock. However, when running the program in an interpreter, the deadlock might first occur after several hundred iterations of the loop in method `run`. Our first goal is to make this code testable in a reproducible way, by replacing the system non-determinism with a deterministic thread scheduler. Our second goal is to reintroduce non-deterministic choices and detect the deadlock.

Figure 7 introduces a replacement of the original multi-threading features with the aspect generated at (1). The

[9]Continuations created by `call/cc` are a kind of bookmarking mechanism of program executions — calling a continuation switches the execution context including the call stack to the context where the continuation was created [8]. When simulating threads with continuations, the concrete program and the thread scheduler are co-routines, and `call/cc` allows switching between both.

```
class Actor
  attr_accessor :other

  def initialize; @mutex = Mutex.new; end

  def run
    loop do
      @mutex.synchronize do other.interact end
    end
  end

  def interact
    @mutex.synchronize do ... end
  end
end

def deadlock
  a1 = Actor.new
  a2 = Actor.new

  a1.other = a2
  a2.other = a1

  t1 = Thread.new(a1) do |a| a.run end
  t2 = Thread.new(a2) do |a| a.run end

  t1.join; t2.join
end
```

Figure 6: Simple code producing a deadlock

aspect intercepts creation of `type1` and instead instantiates `type2` with the original arguments.[10] The aspect is used for replacing `Thread` and `Mutex` at (3) using the strategy initialized at (2), that prevents a further propagation of the aspect over the call stack. The replacement classes are part of the ExploR framework and implement multi-threading with co-routines using `call/cc` context switches.

Each test run is parameterized with a closure used by the scheduler to choose the next executing thread. First, we perform a deterministic approach by using a constant choosing closure at (4), which chooses always the first available thread, so that the program executes deterministically. To achieve our second goal, we remove the comment sign at (5) in order to apply the non-deterministic choice operator for choosing the next thread. In the default configuration, the scheduler makes thread choices at each synchronization point and at thread starting points. This results in 52 different executions of the example, out of which 12 lead to a deadlock, detected by our tool. Our framework does not introduce context switches outside of synchronization calls, which leads to an under-approximation of the original thread behavior. However, more context switches can be inserted either manually or using code instrumentation. To avoid a combinatorial explosion, it is also possible to select a different exploration method (the default is exhaustive depth-first search) or to specify bounds on the execution [16].

5. IMPLEMENTATION

In the following, we briefly discuss our implementation[11] and the underlying architecture.

[10]The `eval` method in Ruby performs dynamic evaluation of a string of code.

[11]The implementation and all examples shown in this work are available at `http://twister.rubyforge.org`

```
    def replace type1, type2
(1)   return Aspect.new do
        around new & name(type1) do
            eval("#{type2}.new *args, &block")
          end
      end
    end

(2) strategy = Strategy.new {[False,False,True]}
    replace_thread = replace 'Thread', 'SimulatedThread'
    replace_mutex = replace 'Mutex', 'SimulatedMutex'

(3) deploy_on Thread, replace_thread, strategy
    deploy_on Mutex, replace_mutex, strategy

    MultiThreadedTestProcessor.new {
(4)   choosing = lambda{ |t| t[0] }
(5)   # choosing = lambda{ |t| choice(t) }
      thread_test choosing do
        main = SimulatedMainThread.new do
          deadlock
        end
      end
    }.run
```

Figure 7: Test harness for multi-threaded programs

5.1 ExploR

Program exploration with ExploR is implemented as a library in Ruby, so that no interpreter extension is necessary. Each test method is passed to ExploR as a block of code that can be executed multiple times. During a set of multiple executions, a data structure representing the choice tree is maintained by the framework. Each call to the choice function represents a node in the choice tree. The first call to the choice function will initialize a new child node in the current choice branch, while a repeated call to the same choice in a subsequent execution will advance to the next choice.

The framework provides a preliminary stopping mechanism for each execution, by the use of context switches with `call/cc`. It is, for example, used in the `assert` function of the framework, which will stop the current execution and switch to the next choice set immediately. This is in contrast to assertions that throw exceptions, which can be catched and shielded by production code. A context switch to the next execution is also used when the framework described in Section 4 is in the deadlock state.

5.2 TwisteR

TwisteR is a dynamic aspect language, implemented as a library in Ruby. In the following, we only briefly summarize implementation issues, since a deeper insight is beyond the scope of this paper. For details we refer the reader to [3].

Code instrumentation and design adaptations are performed using metaprogramming and reflection. A thread-local stack of aspect environments is maintained to control aspect propagation and scoping over the call stack. The Ruby base class `Object` is extended with an aspect environment to allow per object aspect deployment and scoping. TwisteR is designed as a small core language to be extended for specialized domains. Extensions include also new pseudo-keywords for aspect, pointcut and advice definition.

6. RELATED WORK

Explicit state model checkers such as Java PathFinder provide an API supporting non-deterministic choices in concrete program code [18]. Features like backtracking, state matching and partial order reduction avoid revisiting identical executions and reduce the state space. This comes with the cost of an additional level of interpretation by a specialized virtual machine. In our ExploR framework, non-deterministic choice is performed on the same level of interpretation as the analyzed code, but each separate execution is performed from the beginning without detecting similar states. Avoiding an extensive interpreter extension, ExploR is implemented as a compact and extensible library.

Non-deterministic choice provided by explicit state model checkers has been used for generating structurally complex test input, e.g., object graphs [10, 7]. In contrast to our approach, the non-deterministic programs are executed for test generation, while we directly perform testing on the non-deterministic code. We focus furthermore on the design of stubs and mock objects in unit testing with non-deterministic choice.

Static aspect-oriented programming techniques have been applied before in testing to introduce mock objects, access private data or for monitoring purposes [13, 9, 2, 12]. Code is instrumented at compile-time or load-time, so that it is difficult to distinguish between different runtime contexts. Our approach, on the contrary, uses dynamic and per object deployment of aspects to enable more test flexibility and to keep adaptations performed by aspects local to a particular control flow or object reference.

7. CONCLUSION

We presented a new approach to testing with non-deterministic choice and dynamic aspects. We illustrated a compact design of unit tests that subsume a number of different test cases driven by our framework ExploR. Flexible test implementations and integrations were performed using dynamic aspect deployment with the TwisteR aspect language. A combination of non-determinism and dynamic aspects facilitated a flexible approach of testing multi-threaded programs. We believe that both approaches will lead to faster and simpler test development for dynamically-typed languages.

8. REFERENCES

[1] Mocha. `http://mocha.rubyforge.org/`, 2006.

[2] M. Achenbach and K. Ostermann. Engineering abstractions in model checking and testing. In *Proceedings of the Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 137–146, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[3] M. Achenbach and K. Ostermann. Growing a dynamic aspect language in Ruby. In *DSAL '10: Proceedings of the 2010 AOSD workshop on domain-specific aspect languages*. ACM Press, 2010. To Appear.

[4] D. Astels. *Test Driven development: A Practical Guide*. Prentice Hall Professional Technical Reference, 2003.

[5] S. Freeman, T. Mackinnon, N. Pryce, and J. Walnes. Mock roles, not objects. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 246, 236. ACM Press, 2004.

[6] S. Freeman and N. Pryce. *Growing Object-Oriented Software, Guided by Tests*. Addison-Wesley Professional, 2009.

[7] M. Gligoric, S. Khurshid, T. Gvero, V. Kuncak, V. Jagannath, and D. Marinov. Test generation through programming in UDITA. In *ICSE '10: Proceedings of the 32nd International Conference on Software Engineering*, Cape Town, South Africa, 2010. To Appear.

[8] C. T. Haynes, D. P. Friedman, and M. Wand. Continuations and coroutines. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 293–298, New York, NY, USA, 1984. ACM.

[9] R. Jeffries. Virtual mock objects using AspectJ with JUNIT. `http://www.xprogramming.com/xpmag/virtualMockObjects.htm`, 2002.

[10] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *TACAS'03: Proceedings of the 9th international conference on Tools and algorithms for the construction and analysis of systems*, pages 553–568, Berlin, Heidelberg, 2003. Springer-Verlag.

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353. Springer-Verlag, 2001.

[12] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications, July 2003.

[13] N. Lesiecki. Test flexibly with AspectJ and mock objects. `http://www.ibm.com/developerworks/java/library/j-aspectj2/`, 2002.

[14] X. Li, H. J. Hoover, and P. Rudnicki. Towards automatic exception safety verification. In *Proceedings of the 14th International Symposium on Formal Methods*, pages 396–411. Springer, 2006.

[15] T. Mackinnon, S. Freeman, and P. Craig. *Endo-testing: unit testing with mock objects*, pages 287–301. Addison-Wesley Longman Publishing Co., Inc., 2001.

[16] K. Sullivan, J. Yang, D. Coppit, S. Khurshid, and D. Jackson. Software assurance by bounded exhaustive testing. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 133–142, New York, NY, USA, 2004. ACM.

[17] É. Tanter. Expressive scoping of dynamically-deployed aspects. In *Proceedings of the 7th international conference on Aspect-oriented software development*, pages 168–179, New York, NY, USA, 2008. ACM.

[18] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Automated Software Engineering, 2000. Proceedings ASE 2000. The Fifteenth IEEE International Conference on*, pages 3–11, 2000.