

# Growing a Dynamic Aspect Language in Ruby

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

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

## ABSTRACT

We present an approach to embed constructs for aspect-oriented programming with dynamic deployment in Ruby using metaprogramming. The AOP constructs developed in this way facilitate dynamic instantiation of aspects and deployment with expressive scoping strategies. Domain-specific extensions of these constructs make our approach particularly well-suited for domains that require more complex, invasive, or dynamic instrumentation beyond AspectJ-like AOP languages.

## Categories and Subject Descriptors

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

## Keywords

Metaprogramming, Ruby, Domain-specific Aspect Languages, Dynamic Aspect Deployment

## 1. INTRODUCTION

Different domains like testing, logging, debugging, design by contract, or security would benefit from an expressive and flexible dynamic deployment of aspects. Since all domains have different needs, it is desirable to separate their concerns into different aspect languages. While, for instance, integration testing would benefit from managing different class implementations in different scopes [3], a design by contract specification language would require the definition of pre- and postconditions of methods [14].

In earlier work we analyzed the applicability of AspectJ for testing and model checking of Java programs [3]. It turned out that AspectJ is not flexible enough for using, e.g., different implementations of a class at runtime to serve as mock objects, and that the global scoping of aspects hinders local testing or the usage of different aspects in different contexts. This is in line with other previous work calling for AOP support in integration testing [11, 9].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

AOSD '10 Rennes and Saint Malo, France  
Copyright 2010 ACM X-XXXXX-XX-X/XX/XX ...\$10.00.

We therefore implemented a core aspect language, TwisteR, as an internal domain-specific language (DSL) for Ruby to be extended with domain-specific extensions (DSX) for separate domains. Ruby is a widely used dynamically typed language [16], providing expressive metaprogramming features. Techniques like open classes and dynamic evaluation enable embedding a language like TwisteR as an internal DSL without the requirement of an interpreter extension. Besides the dynamicity inherent in Ruby itself (e.g., no static typing constraints), our main weapon of adding flexibility is by a powerful notion of dynamic aspect deployment. Dynamic aspect deployment is a well-known approach to increase the flexibility of aspects [4, 8]. The most general dynamic deployment approach aware to the authors has been described by Tanter [15], and this approach has been adopted for TwisteR.

The contributions of this work are as follows:

- We present the extensible dynamic aspect language TwisteR — an experimentation platform for deploying aspects with dynamic scoping strategies
- We discuss a step-by-step extension of the language: first with AspectJ-like pointcuts and advice, including efficient `cflow` simulation, then with dynamic DSXs in domains like design by contract and integration testing

The remainder of this paper is organized as follows: Section 2 presents our language core and describes the deployment strategies approach. Section 3 discusses technical details of the implementation, while Section 4 presents extensions of our language. We discuss related work in Section 5 followed by our conclusion.

## 2. TwisteR DESIGN AND FEATURES

In the following we introduce TwisteR, accompanied by an introduction of Ruby features and syntax relevant to this work. Thereafter we illustrate the *deployment strategies*, which are adopted by our implementation.

### 2.1 Defining Aspects

Similar to Aspect Scheme [8], pointcut and advice in our aspect language are so called blocks, which are anonymous method bodies that depend on the context of their definition (e.g., variables from that context can be used and are bound to their value at definition time). Blocks are surrounded by curly braces or a `do`, `end` pair. A pointcut is expected to return `true` or `false`, defining the matching join points, while an advice contains the code to be executed in case of a match. Aspects are containers for pointcut/advice

pairs. They are runtime objects in our approach, i.e., they have the properties of first-class values and can also be dynamically generated and instantiated. The aspect definition depends on variables from the surrounding context, so that it can handle method and class names first known at runtime, which is suitable for applications creating methods on the fly. The following code shows the instantiation of an aspect, which passes its definition as an anonymous block to the constructor:

```
tracing = BaseAspect.new do
  around pc{jp.type=='execution'} do
    puts "Tracing #{jp.name}"
    proceed
  end
end
```

The aspect matches method executions, writes some output, and calls `proceed`, which in turn calls the advised method or the next matching around advice. Within the interpretation context of the aspect’s block, keywords like `around` or `pc` can be used to define an around advice or a pointcut respectively. The pointcut defined with `pc` is a Ruby closure, which is a block that can also be assigned to variables or method parameters. Within the interpretation context of the block given to `pc`, the `jp` keyword gives access to various data corresponding to the dynamic join point. Note that the mere aspect definition does not have any effect, since it is not deployed yet.

## 2.2 Dynamic Deployment

Static aspects like in AspectJ [10] have a global scope, which has to be refined for each pointcut if local applications are needed. As a consequence, pointcut definitions are scattered within one aspect, such that a change of scope of the entire aspect leads to several modifications [15].

Dynamic deployment of aspects has been proposed for several languages, e.g., in CaesarJ [4] with the `deploy a {...}` statement, that deploys an aspect in the context of the following block, or in Aspect Scheme [8], with the `around` and `fluid-around` statements. As shown by Tanter [15], these deployment mechanisms are fragile to software evolution, like the application of refactorings.

We adopted therefore Tanter’s deployment strategies [15] in our language. An aspect `asp` is deployed in the scope of a block using the `deploy asp, str do ... end` function, where the deployment strategy `str` is a triple of strategy functions  $\{c, d, f\}$ . These functions control the propagation of aspects, which we informally explain in the following. Formal definitions can be found in [15].

### 2.2.1 Strategy Functions

An aspect environment is a list of active aspects and their associated deployment strategies. We assume an aspect environment  $\mathcal{A}$  being associated with each dynamic join point (e.g., at method calls or for object creation). At a join point, the environment is filtered by applying one of the functions  $\{c, d, f\}$ , which take the join point as an argument and return a boolean.

Function  $c$  controls propagation of the aspect over the call stack ( $c$  like call stack propagation). If a method call is evaluated in an aspect environment  $\mathcal{A}$ , the body of that method is executed in an environment obtained by filtering the  $(asp, str)$  pairs in  $\mathcal{A}$  applying the  $c$  function of each strategy `str` to the dynamic join point corresponding to the

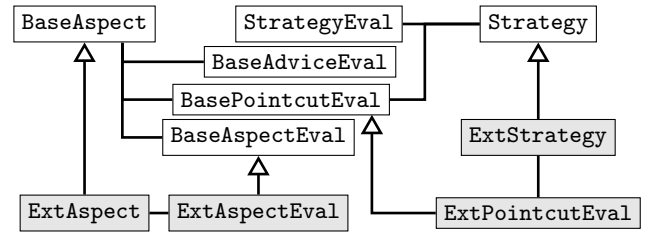


Figure 1: Base classes and possible extensions

method call.

Function  $d$  allows the dynamic scope of an aspect to exceed the scope of the block where it got deployed ( $d$  like delayed evaluation propagation). When an object is created, the aspect environment of the creation-site (i.e. where the constructor is called) is attached to the object. If one of the object’s instance methods is called (in- or outside the deployment block), the object’s aspect environment is merged with the one propagated over the stack to evaluate the method’s body. However, not all aspects are attached to an object, since the aspect environment is filtered by  $d$  applied to the object creation join point.

Function  $f$  serves as a filter to the whole aspect ( $f$  like filter), so that several advice statements of one aspect are controlled accordingly. Before processing pointcuts, the aspect environment is filtered by applying  $f$  to the join point considered for weaving.

### 2.2.2 Deploying Aspects

In the following, we deploy the tracing aspect from Section 2.1. The strategy `TFX` uses two constant closures `True` and `False`, which return `true` and `false` for  $c$  and  $d$  respectively. The filter  $f$  applies the aspect only to methods with a name different than `delegate`:

```
def access x; puts "Accesssing: #{x}"; end
def delegate x; access x; end
TFX = Strategy.new{[True, False, pc{jp.name!='delegate'}]}
deploy tracing, TFX do delegate :some_resource end
# Outputs:
# Tracing access
# Accesssing: some_resource
```

The strategy’s block is interpreted similar to aspects (e.g., providing keywords like `pc`, `True` and `False`) and is expected to return the strategy functions as a triple of closures. Each strategy function is evaluated like a pointcut, so that the join point to which the function is applied, can be accessed with the `jp` keyword. In the remainder of this paper, we will reuse the constant functions using self-explanatory strategy names like `TFT` or `TTT`.

## 2.3 Extensibility

Figure 1 gives an overview of the main classes used for extensions. The `BaseAspect` and `Strategy` class serve as superclasses of all specialized aspect language implementations. When initialized, they take their definition as a block, as shown in Section 2.1 for aspects. The block contains constructs of the DSL mixed with Ruby code. Similar to the approach in [6], the block is evaluated in the context of an evaluator instance, resolving language constructs as pretended property accesses and method calls. E.g., `around` is

a method defined in `BaseAspectEval`, taking a pointcut as an argument and the advice as a block, storing this block in the advice list of the aspect.

There are different interpretation contexts for the aspect's block, pointcut, advice, the deployment strategy and the strategy functions. These contexts can be extended with specialized DSXs by subclassing the corresponding evaluator classes shown in Figure 1, adding methods, constants or metaclasses. The grey classes exemplify the inheritance hierarchy of a possible extension.

### 3. TECHNICAL DETAILS

In the following we give an overview of Ruby's metaprogramming features facilitating our implementation<sup>1</sup>, which is described below.

#### 3.1 Ruby Metaprogramming

The main metaprogramming features used in our implementation are open classes combined with dynamic evaluation, as well as internal callback methods, notifying creation of methods and classes.

Open classes allow reopening of a class body to redefine or add methods, constructors, or declarations. The `alias` statement allows storing an old method definition under a different name, so that it is still accessible after redefining the original method.

Using dynamic evaluation, strings or blocks of code can be evaluated at runtime. The evaluation can be performed in different contexts, at class- or instance-level and with different variable-bindings to interpret keywords as pretended method calls.

#### 3.2 Instrumentation and Aspect Propagation

In the current implementation, dynamic join points are generated for the execution of every method body of user-defined code (including class creation by calling the `new` method of a class). The callback methods `method_added` of class `Module` and `inherited` of class `Class` keep track of newly defined methods and classes. Using a combination of dynamic evaluation and open classes, each location of a dynamic join point is instrumented with a method-alias, while the original method is redefined. The redefined method body maintains a stack of aspect environments, aspect propagation and weaving corresponding to the definitions in Section 2.2. A call to the aliased method is stored as a closure, which is either called manually with `proceed` in matching around advice or automatically, if no advice matches.

#### 3.3 Efficiency

Our approach is less efficient than aspect languages that statically weave advice for matching join points. Different overheads can appear at (1) aspect definition time, (2) method definition time, and (3) method call time. The parsing of the aspect definition at (1) did not show any significant performance loss and method instrumentation is only performed once at (2). We did not investigate yet if this impacts systems that dynamically create a huge amount of methods and classes on the fly. At (3), propagation and dynamic weaving is performed. Empty aspect environments are handled efficiently, so that a possible slow down affects only

<sup>1</sup>TwisteR and all examples shown in this work are available at <http://twister.rubyforge.org>

these dynamic contexts of a program where aspects are deployed. The time for weaving and propagating a non-empty aspect environment is an inherent factor of the deployment strategies approach and depends on the configuration of the strategy functions and the number of aspects.

## 4. GROWING THE LANGUAGE

In the following, we show how to extend the core language with specialized DSXs to obtain a simple AspectJ-like language, a simulation of `cflow`, and specializations in the domains design by contract and testing.

### 4.1 Building AspectJ-like Aspects

AspectJ has a rich pointcut and advice language and is based on compile-time or load-time weaving [10]. To emulate AspectJ-like pointcut and advice in our dynamic aspect language, the base aspect's evaluator is extended as follows:

```
class AspectEval < BaseAspectEval
  def execution; pc{type=='execution'}; end
  def new; pc{ type=='new'}; end
  def name n; pc{ name==n}; end
  def target_type t; pc{ target.class.name==t}; end

  def before pc, &advice
    around pc do
      eval_advice(advice, jp)
    proceed
  end
end
end
```

New pointcut keywords allow a more convenient formulation of pointcuts. Specialized advice is based on the built-in `around advice`. The call to `eval_advice` will evaluate the block given to the advice in the context of the advice evaluator attached to `Aspect`. Both advice and pointcut evaluator (not shown here) are extended with some syntactic sugar to allow, e.g., writing `name` instead of `jp.name`. Other runtime information not stored at `jp` could be retrieved using dynamic evaluation on the receiver `target`.

### 4.2 Simulating `cflow` with Local Deployment

To obtain more expressive dynamic scoping of aspects, languages like AspectJ provide a `cflow` pointcut. In [13], several approaches to determine matching join points are presented. The naive approach can be implemented as a pointcut that accesses the call stack via the `parent` field of the join point. However, iterating over the call stack to determine `cflow` is very inefficient.

To simulate the efficient approach presented in [13], we deploy an aspect which matches the desired control flow and in turn locally deploys the `cflow`-controlled aspect in the appropriate scope. Figure 2 shows the implementation of the `deploy_cflow` method, which takes a method name `m` and an aspect `a` and simulates adding a `cflow(m)` pointcut to the deployment strategy of `a`. Using the `deploy` method from Section 2.2, at (3), the `deploy_cflow` method deploys `cflow_aspect` defined at (1), that instruments the matching method `m` by locally deploying the aspect `a` only in the scope of `m`, so that determining matching join points is done in constant time. The propagation strategy defined at (2) limits the dynamic extent of `a` to the next occurrence of `m` to prevent a duplicate deployment.

The second part of Figure 2 demonstrates the application of `deploy_cflow` using pointcut and advice introduced in

```

# Function simulating a cflow pointcut
def deploy_cflow m, a
(1)  cflow_aspect = Aspect.new do
      around execution & name(m) do
(2)    s = ExtStrategy.new {[name(m), False, True]}
        deploy a, s do proceed end
      end
(3)  end
      deploy cflow_aspect, TFT do yield end
    end
# Client code for tracing
def foo; puts "foo"; end
def bar; puts "bar"; foo; end
def main
  puts "Calling foo (no match):"
  foo

  puts "Calling bar (matches foo):"
  bar
end
# Aspect that traces method foo
(4) a = Aspect.new do
      before execution & name('foo') do
        puts "Tracing #{name}"
      end
    end
# Deploy the aspect in the cflow of 'bar'
deploy_cflow 'bar', a do main end

```

Figure 2: Simulating cflow using deploy

Section 4.1. The tracing aspect at (4) traces the execution of `foo` only in the control flow of `bar`.

In AspectJ it is also possible to bind information of the calling context of the join point matching the `cflow` pointcut to a variable. In our example, this would be the access to a possible argument of `bar` in the advice matching `foo`. Since there might be different occurrences of `bar` in the call stack of `foo`, [13] proposes a stack of corresponding variable bindings, which could be simulated also in our example in Figure 2 around the `proceed` call of method `m`.

### 4.3 On Building a Design by Contract DSX

*Design by contract (DBC)* is a form of programming with precise and verifiable interfaces including preconditions, postconditions and invariants [14], a paradigm, also beneficial for a dynamically typed language like Ruby, which provides little static guarantees. Our focus is to build a practical and expressive DSX to our language to formulate reusable preconditions, postconditions and invariants as aspects, being deployed in the development and test phase of a program. We only discuss a small subset of such a DSX, since a full specification language is out of the scope of this paper.

The first part of Figure 3 shows a minimal implementation, as a subclass of the aspect evaluator of Section 4.1, with a construct for preconditions, `dbc_require` (1), and a predicate `forall_args` (2) checking a condition for all arguments of matching functions. The `dbc_require` construct takes a list of pointcuts and conditions in Ruby hash syntax. A condition is a closure evaluated on the same evaluator instance. An exception is thrown if it does not hold. The `forall_args` predicate takes a block that checks one argument, and generates a closure that checks all arguments.

The second part of Figure 3 demonstrates the application of the DBC aspect language at (3), building a non-nil pre-

```

# Design by contract extension
class DbCEvaluator < AspectEvaluator
(1)  def dbc_require cond
      cond.each{|pc, c|
        before pc, do
          unless instance_eval(&c)
            raise "Violation at #{name}"
          end
        end
      }
    end
(2)  def forall_args
      pc{args.inject(true){|res, arg| (yield arg) && res}}
    end
# The client code to be extended with a contract aspect
class UserInterface
  def print arg; end
  def debug arg; print nil; end
end
# Defining and deploying a contract
(3) dbc = DbC.new do
      dbc_require target_type('UserInterface') =>
        forall_args{|arg| arg != nil}
    end
(4) strategy = ExtStrategy.new {[name('debug'), False, True]}
(5) deploy dbc, strategy do
      t = UserInterface.new
      t.print nil
      t.debug 'test'
    end

```

Figure 3: Subset of a Design by Contract DSX and its application

condition for all arguments of functions of a specified type (here class `UserInterface`). The aspect specifying the requirements is deployed at (5) with the strategy function at (4), which aborts stack propagation below method `debug`. The call to `print` with `nil` will cause a precondition violation exception, but once it is fixed, the call to `print` from within method `debug` does not raise an exception, since it is out of the scope of this aspect. The existing pointcut language is reused and mixed with the design by contract constructs, e.g., `target_type` to determine matching types. Currently, the stack trace of the exception reveals AOP details not relevant to the programmer — it remains an open problem to conceal them.

### 4.4 Class Substitution in Integration Testing

For statically typed programming languages, there are several approaches for managing different class versions at runtime, like Classboxes [5] or the Keris module system [18]. However, dynamic scoping cannot be expressed in these languages. In integration testing, the dynamic scope of a mock object [12] might be limited or there might be different mock-implementations or test stubs needed at runtime [3]. Ruby, being a language with duck-typing, trivially provides the flexibility to exchange classes. However, since there is no means to specify the dynamic scope of such a modification, we combine class exchange at instantiation-time with the deployment strategies approach.

The following example shows the `SubstituteEvaluator`, used by the `Substitution` aspect (not shown here) to control the dynamic extent of class exchange. The second part of

the example shows the substitution of IO components for testing `SystemUnderTest` in isolation:

```
class SubstituteEvaluator < AspectEvaluator
  def substitute type1, type2
    around new & name(type1) do
      eval("#{type2}.new *args, &block")
    end
  end
end
subst = Substitution.new do
  substitute 'DatabaseHandler', 'DatabaseMockClass'
  substitute 'FileReader', 'FileReaderStub'
end
s = ExtStrategy.new {[!name('log'), False, True]}
deploy subst, s do SystemUnderTest.run end
```

The deployment strategy performs a stack propagation excluding the logging mechanism of the system under test.

## 5. RELATED WORK

Deployment strategies have been implemented in Scheme [15] and in JavaScript [17]. Similar, but less expressive approaches like [8, 4] were already discussed in Section 2.2.

Our language embedding using pretended method calls is similar to the Groovy-approach from [6]. However, that approach is orthogonal, since it focuses on static weaving using an AspectJ-extension. While advice can be dynamically linked, the dynamic scope of aspects is limited to the expressiveness of AspectJ. In our approach, evaluator extensions facilitate the introduction of new domain-specific language constructs, while a clearly defined Meta-Aspect Protocol as in [7] remains future work. In that approach, a meta-language-interface facilitates modification and extension of pointcut matching and advice execution semantics. Adaptation of the semantics of aspect propagation, however, is not explicitly discussed.

Aquarium [2] and AspectR [1] are AOP-implementations for Ruby, focusing on the simulation of AspectJ-like pointcut and advice, Aquarium being the more elaborated approach. Unlike our approach, Aquarium performs only static weaving at aspect-definition-time, which does not facilitate any dynamic scoping. The aim of Aquarium is orthogonal to our approach: It provides an efficient and expressive static pointcut language, while we focus on an extensible core language, facilitating dynamic scoping of aspects.

## 6. CONCLUSION AND FUTURE WORK

We presented TwisteR, an aspect language for Ruby, implementing dynamic deployment strategies. We demonstrated our approach on building extensions to the language core in several domains. In our future work we plan to enhance composability of our `cflow` simulation by integrating nested dynamic deployment into a homogenous pointcut language. We further plan to optimize our approach by separating the dynamic and static concerns of pointcuts. A further development of our design will lead to expressive extensions in domains requiring dynamic deployment.

## 7. REFERENCES

- [1] Aspectr. <http://aspectr.sourceforge.net/>, 2002.
- [2] Aquarium. <http://aquarium.rubyforge.org/>, 2009.
- [3] 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.
- [4] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. *An Overview of CaesarJ*. 2006.
- [5] A. Bergel, S. Ducasse, and R. Wuyts. *Classboxes: A Minimal Module Model Supporting Local Rebinding*, pages 122–131. 2003.
- [6] T. Dinkelaker and M. Mezini. Dynamically linked domain-specific extensions for advice languages. In *DSAL '08: Proceedings of the 2008 AOSD workshop on Domain-specific aspect languages*, pages 1–7, New York, NY, USA, 2008. ACM.
- [7] T. Dinkelaker, M. Mezini, and C. Bockisch. The art of the meta-aspect protocol. In *AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 51–62, New York, NY, USA, 2009. ACM.
- [8] C. Dutchyn, D. B. Tucker, and S. Krishnamurthi. Semantics and scoping of aspects in higher-order languages. *Sci. Comput. Program.*, 63(3):207–239, 2006.
- [9] R. Jeffries. Virtual mock objects using AspectJ with JUNIT. <http://www.xprogramming.com/xpmag/virtualMockObjects.htm>, 2002.
- [10] 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.
- [11] N. Lesiecki. Test flexibly with AspectJ and mock objects. <http://www.ibm.com/developerworks/java/library/j-aspectj2/>, 2002.
- [12] T. Mackinnon, S. Freeman, and P. Craig. *Endo-testing: unit testing with mock objects*, pages 287–301. Addison-Wesley Longman Publishing Co., Inc., 2001.
- [13] H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In *Compiler Construction, volume 2622 of Springer Lecture Notes in Computer Science*, pages 46–60. Springer, 2003.
- [14] B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.
- [15] 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.
- [16] D. Thomas and A. Hunt. *Programming Ruby: the pragmatic programmer's guide*. Addison-Wesley Longman Publishing Co., Boston, MA, USA, 2000.
- [17] R. Toledo, P. Leger, and É. Tanter. AspectScript: Expressive aspects for the Web. In *Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development*, Rennes and Saint Malo, France, Mar. 2010. ACM Press. To Appear.
- [18] M. Zenger. Keris: evolving software with extensible modules: Research articles. *J. Softw. Maint. Evol.*, 17(5):333–362, 2005.