A Meta-Aspect Protocol for Developing Dynamic Analyses

Michael Achenbach[1] and Klaus Ostermann[2]

[1] Aarhus University, Denmark, `ma@cs.au.dk`
[2] University of Marburg, Germany, `kos@informatik.uni-marburg.de`

Abstract. Dynamic aspect-oriented programming has been widely used for the development of dynamic analyses to abstract over low-level program instrumentation. Due to particular feature requirements in different analysis domains like debugging or testing, many different aspect languages were developed from scratch or by extensive compiler or interpreter extensions. We introduce another level of abstraction in form of a meta-aspect protocol to separate the host language from the analysis domain. A language expert can use this protocol to tailor an analysis-specific aspect language, based on which a domain expert can develop a particular analysis. Our design enables a flexible specification of the join point model, configurability of aspect deployment and scoping, and extensibility of pointcut and advice language. We present the application of our design to different dynamic analysis domains.

1 Introduction

Many dynamic analyses make use of program instrumentation tools to transform the abstract-syntax tree (AST) of analyzed code. This leads to a tight coupling between low-level language and analysis design. A designer needs expert knowledge of the language specification, evolution of the language makes analyses brittle, and the analysis code base includes many instrumentation details [5].

Aspect-oriented programming (AOP) has been used to overcome these problems [17, 25]. AOP allows high-level abstractions of program instrumentation using pointcut and advice, and facilitates rapid design of small analysis aspects, decoupled from language details. However, different dynamic analyses require different aspect language features and instrumentation techniques. A debugging tool might require a fine-grained instrumentation level (e.g., statement-based) [19], while a memory profiler might need only object-allocation instrumentation [24]. Often, domain-specific aspect languages (DSALs) are built from scratch or by extensive compiler extensions to fit particular requirements. A general-purpose AOP language that covers all possible features, however, induces unnecessary overhead and requires broader expert knowledge than a DSAL.

In object-oriented languages, metaobject protocols (MOPs) have been developed to access and adapt the language semantics from the programming level of the language itself [14]. In AOP, meta-aspect protocols (MAPs) have been suggested, where parts of the language semantics are like in MOPs controlled through a concrete interface [11]. This enables domain-specific extensions (DSX) without the requirement of a new compiler or interpreter. However, there exists no MAP known to the authors that covers the requirements of dynamic analyses.

We present a MAP that focuses in particular on the requirements of dynamic analyses in the setting of dynamically typed languages. It is based on a load-time program transformation that inserts hooks for dynamic aspect weaving and scoping. The join point model is based on a user-defined AST-transformation, so that every syntactic element can be included as a join point. The language interface facilitates the configuration of different dynamic deployment methods (global, per block, per reference) and scoping mechanisms (stack/reference propagation) [4, 6, 21], and the extension of pointcut and advice [1].

We present two DSXs based on our protocol focusing on the requirements of the domains debugging and testing, respectively. Based on these extensions, we develop two exemplary analysis aspects with a small code base.

The contributions of this work are as follows:

- We discuss several dynamic analyses that build on AOP. We point out particular dynamic AOP features that are required in certain analysis domains.
- We present a dynamic meta-aspect protocol that provides direct access to the aspect language semantics at runtime and allows the configuration of domain-specific extensions for particular dynamic analysis domains.
- We provide an implementation of our extensible language and evaluate its applicability by developing language extensions and analysis aspects for the domains debugging and testing.

We chose Ruby as implementation language, but the design could be applied to other dynamic languages like Groovy as well. We intend our MAP to be used for prototyping and developing analyses in domains like debugging and testing, which are not limited by strict performance requirements, since dynamic AOP comes with the cost of a certain runtime overhead.

The remainder of this paper is organized as follows: We analyze the requirements of dynamic analyses in Sec. 2 and discuss related AOP approaches. Section 3 presents the design of our meta-aspect protocol for dynamic analyses. We evaluate our protocol in Sec. 4 on two example applications, followed by our conclusion.

2 Motivation

In the following, we analyze the requirements of dynamic analysis regarding AOP, in order to build a flexible abstraction layer over program instrumentation. We then discuss related approaches that either lack flexibility or generality.

2.1 Interception Requirements of Dynamic Analyses

Dynamic analyses vary widely in their interception requirements. We structure the resulting design space in four dimensions: Join point model, aspect deployment, aspect scope, and pointcut and advice language.

Join Point Model Different analyses require different kinds of access to the program structure or execution context in form of join points.[3] To analyze performance and relation of method calls or to analyze memory consumption, method execution and object allocation join points are typically sufficient [24]. These requirements are met by general-purpose AOP languages like AspectJ [13]. However, more advanced monitoring and debugging tools require instrumentation on the level of basic blocks [5] or statements like assignment [17]. The debugging approach of [19] also introduces line number join points. Data-race detection is based on field access join points [7]. No general-purpose AOP language known to the authors meets all these requirements.

Aspect Deployment The entity on which an aspect is deployed differs among many dynamic analyses (e.g., on the whole program, on objects or methods). Bodden and Stolz suggest dynamic advice deployment (with a global deploy/undeploy functionality) and per object deployment as in Steamloom [6] to optimize temporal pointcuts over execution traces [8]. Toledo et al. also deploy security aspects in the dynamic scope of application objects [23]. In earlier work, we used deployment on a block of code like in CaesarJ [4] to separate design changing aspects used in different test cases [2]. Deployment on a block is particularly powerful in combination with expressive aspect scoping [21].

Aspect Scope Dynamic analyses often require fine-grained control over the scope in which an aspect is active to determine what should be and what should not be analyzed. E.g., omniscient debuggers (that can step backward in time) rely on vast execution traces. Tanter suggests therefore a notion of partial traces defined by an expressive scoping strategy [22]. Toledo et al. embed web application code into the scope of a security aspect that monitors access attempts [23]. A so called *pervasive* scope ensures that neither method calls nor references escape the aspect.

Pointcut and Advice Language Allan et al. introduce binding of free variables in pointcut expressions for trace matching [3]. Dinkelaker et al. integrate a 3-valued logic language into the advice language [10], which is advantageous for analyses that reason with boolean abstractions at runtime like Java PathFinder [26]. Aspects that augment a program for test generation could be extended with domain-specific constructs for non-deterministic choice [12]. The possibility to specify new keywords for pointcut and advice enables shorter, analysis-specific pointcut and advice definitions [1].

Our survey shows that certain classes of analyses require different AOP features. General-purpose AOP does not solve the problem. If it covered all necessary AOP

[3] A join point is a point in the program text (static) or in the execution (dynamic), where an interception with aspects can take place. A join point model defines the set of possible join points and also the kind of interaction, e.g., if and how the program state or control-flow can be changed at the join point and how aspects interact.

features of all analysis domains, it would induce a lot of performance overhead for some domains. If it was specialized, some domains would lack important features. Furthermore, the more features are covered, the more expert knowledge is required by the analysis developer. A DSAL, in contrast, allows for domain-specific abstractions of the respective analysis domain.

To avoid building a new DSAL for each class of dynamic analysis from scratch, we need another level of abstraction between language and analysis domain. This will allow a language expert to tailor a domain-specific extension that covers the features of a certain analysis type like debugging. Based on the DSX, a domain expert can rapidly prototype a dynamic analysis that is decoupled from host language details.

2.2 Related Work

There have been previous works on extensible AOP languages, but none of them meets the requirements for dynamic analyses in a satisfactory way. A meta-aspect protocol (MAP) has been developed for the dynamic language Groovy [11]. The join point model is configurable, based on the one of AspectJ [13], which, as illustrated above, is not sufficient for fine-grained instrumentation, e.g., on basic block level. The MAP facilitates both static and dynamic deployment, but does not provide configurable and expressive dynamic scoping concepts. There are also approaches that extend AspectJ by modifying its sources to include more fine-grained join points [9].

Javana is an instrumentation system for building customized dynamic analysis tools for the JVM [15]. The domain-specific Javana language, provides an AOP-like join point model over virtual machine events like object allocation, class loading, or method compilation. Unlike in our approach, the set of events and the language are fixed and can neither be extended nor reduced.

There are specialized approaches for particular analysis domains. Nusayr et al. suggest an AOP framework for runtime monitoring with basic block join points and pointcuts over time and space [17]. Nir-Buchbinder and Ur present a framework for concurrency-aware analysis tools [16]. Binder et al. develop profiler aspects based on the @J language, that supports basic block join points and inter advice communication [5]. Rakesh develops debuggers with line number pointcuts [19]. All these approaches lack generality by implementing fixed sets of features, focusing on the respective analysis domains.

3 The Meta-Aspect Protocol

A meta-aspect protocol (MAP) gives access to the aspect language semantics through a concrete interface on the program level [11]. From the discussion in Section 2.1 we can derive that a MAP for dynamic analysis requires variability in the following dimensions: We need a flexible join point model that allows every syntactical element to be a potential join point. We require different dynamic deployment methods and expressive dynamic scoping. Finally, pointcut and advice language should be extensible with domain-specific constructs.

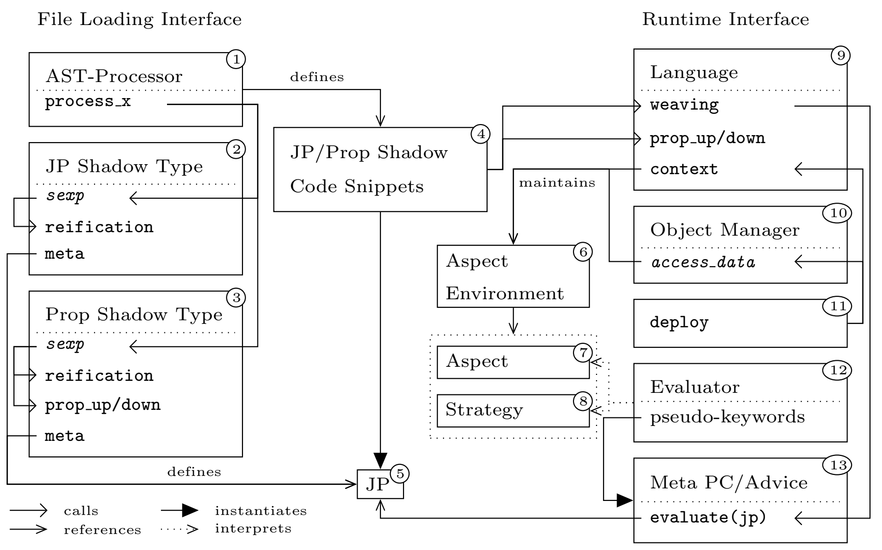

Fig. 1. Overview of the language interface and the main components. Courier fonts refer to actual source methods. Methods in italics are non-modifiable.

We provide a language interface that is instantiated with configurations and extensions of the aforementioned dimensions. We give an overview of the interface and the main components in Sec. 3.1. The join point model is defined by an AST transformation, which we explain in Sec. 3.2. Section 3.3 presents our extensible pointcut and advice language. We illustrate instantiations of different dynamic deployment methods in Sec. 3.4 and demonstrate the configurability of our language w.r.t. expressive aspect scoping in Sec. 3.5.

The features of all four dimensions can be selectively combined. We will instantiate distinguished sets of features to develop two example applications in Sec. 4. The implementation and all examples shown in this work are integrated into the TwisteR project, available at `http://twister.rubyforge.org`.

3.1 Language Interface

Figure 1 gives an overview of the language interface and the main components. We will briefly discuss each and go into more detail in the following sections. The interface is separated into a file loading and a runtime interface. The first comes into play when client code is loaded. The second contains callback methods and extensible classes used at runtime. Loaded code is transformed by a custom AST processor (1), which inserts join point and propagation shadows (4). A join point shadow is a code snippet placed at join points. It guides advice interaction through the `weaving` callback (9). Propagation shadows are code snippets that maintain the scope of aspects. They embed the original code at the point of insertion (e.g., a method call) and configure aspect propagation before and after the embedded code with the `prop_up/down` methods. These methods exist once globally (9) and once for each propagation shadow type (3).

Aspects (7) are associated with scoping strategies (8) and stored in an aspect environment (6). The environment is maintained via the global language interface (9) or through an object manager (10), which associates data with arbitrary runtime objects using reflection. At runtime, join and propagation points are reified through meta objects (5), on which context data is accessible. They are evaluated by meta objects for pointcut and advice (13), guided by the `weaving` callback. Pointcut and advice are instantiated through extensible evaluators (12), which interpret aspects and scoping strategies, for enabling the embedding of domain-specific pointcut and advice languages. Finally, aspect deployment is performed by a user-defined method at (11), which modifies the corresponding aspect environment.

3.2 AST-Based Join Point Model

We represent the AST of a program using S-expressions known from LISP. The join point model of a custom AOP language is defined by a transformer on this expression. The custom S-expression processor is part of the language interface and augments particular AST nodes with join point and propagation shadows for dynamic weaving and scoping.

S-expressions S-expressions are nested list-based data structures. In our implementation, we use external libraries for parsing code and translating S-expressions back to code written in Ruby [18,20]. Each S-expression is represented by a function call `s(...)` listing nested elements as arguments. The list consists of a type symbol, followed by nested S-expressions or primitives like symbols, strings or numbers. We chose the S-expression representation dependent on the external libraries – transforming it into an AST data structure would be straightforward.

Processor for Join Point Generation We distinguish two types of join point shadows: the first type encloses a piece of code, which we will call *interceptor* shadow, the second type only attaches statements to a sequence of code, which we will call *companion* shadow. While the first can change the entire control flow, the second can still access and change the state of the enclosing object.

At load-time, an AST-transformation, defined by a custom S-expression processor (see Fig. 1 (1)), is applied to the code. The transformer performs a traversal over the AST and calls visitor methods for each node type to insert join point shadows. Each join point type is represented by a singleton class (see Fig. 1 (2)) that builds the corresponding shadows. At runtime, each join point is represented by a reified meta object (see Fig. 1 (5)), accessible by pointcut and advice.

Figure 2 illustrates an S-expression processor that integrates method-execution and if-condition join points, which we will need for our testing application (see Sec. 4.1). The class `CustomProcessor` implements visitor methods `process_x` for each node type `x` that needs augmentation with join point shadows. The

```
    class CustomProcessor
        < JoinPointProcessor
(1)   def process_if(exp)
       oldc = process(exp.shift)
       newc = AroundIfCond.sexp(oldc)
       return s(:if, newc, process(exp.shift),
           process(exp.shift))
      end
      def process_defn(exp)
       name = exp.shift
(2)    ast_context[:name] = name
       args = process(exp.shift)
       old = process(exp.shift)
       before = BeforeMExec.sexp
(3)    cap = capture(:result, old)
       after = AfterMExec.sexp(capture)
(4)    newb = s(:block, before, cap.sexp,
           after, cap.var)
       return s(:defn, name, args, newb)
      end
    end
```

```
(5) class AroundIfCond
        < InterceptorShadowType
     def self.reification ast_context
      {:modifier=>:around}
     end
     def self.meta
      IfConditionJoinPoint
     end
    end

(6) class BeforeMExec
        < CompanionShadowType
     def self.reification ast_context
      {:name=>ast_context[:name],
        ...}
     end
     def self.meta
      MethodExecutionJoinPoint
     end
    end
```

Fig. 2. Custom join point processor

singleton classes that create the join point shadows are defined at (5) and (6) (`AfterMExec` not shown here). At (1), we augment the `if`-expression condition with an interceptor shadow that embeds the original code. We iterate over the subexpressions using `exp.shift` and process each recursively.

At method definitions (2), we add the actual method name to the context stack, which stores context information of each AST node and its parents. The original method body is transformed to capture its result in a fresh local variable at (3). The join point shadows are inserted into a sequence before and after the original method body at (4). The method evaluates to the captured result of the original block. Connecting the capturing class and `AfterMExec` enables access to the result through the reified meta join point.

The classes at (5) and (6) differ in their join point reification. The method `reification` defines how to reify the dynamic context at runtime and specifies how to instantiate the meta object that represents the join point. E.g., the method execution join point provides access to the name of the executing method, retrieved from the context stack. Each join point automatically provides access to the self reference of the surrounding object.

Hooks for Weaving Each join point shadow defines the interaction with pointcut and advice. As a default, interceptor shadows introduce *around* advice application and embed the original code to be called if no advice can be applied or if the `proceed` method is called. Companion shadows introduce advice like *before*

```
tracing = Aspect.new do
  before pc{jp.type == :execution} do
    puts "Tracing #{jp.name}"
  end
  after ...
  around pc{jp.type == :if_cond} do
    puts "Cond => #{res = proceed}"
    res
  end
end
```

Fig. 3. Simple tracing aspect

and *after* at discrete points in the program, so that the original control-flow is not modified.

3.3 Pointcut and Advice Language

In earlier work, we developed an extensible pointcut and advice language that we adopted in the current approach [1]. Figure 3 shows the instantiation of a simple aspect that traces method execution and condition evaluation. Aspects are first-class values, their definition is passed as a closure (between `do..end`) to the constructor (called by `Aspect.new`). The closure is interpreted on an extensible evaluator object (see Fig. 1 (12)) on which pseudo-keywords like `pc` are resolved as pretended method calls and property accesses [10]. Behind every keyword is a meta class that represents the construct, e.g., each pointcut is reflected by a meta pointcut with an evaluation function that takes the reified join point as an argument and returns `true` or `false` (see Fig. 1 (13)). The meta classes implement also operators that allow their composition and syntactic sugar (like the operators &,| and ! to compose pointcuts). The `before`, `around`, and `after` pseudo-keywords specify advice that will be executed at matching join points. The pseudo-keyword `jp` gives access to the reified meta join point. In [1], we presented extensions of this approach, e.g., the introduction of new pseudo-keywords, which enables an AspectJ-like syntax, or the simulation of `cflow`. Extensions could also comprise constructs that embed temporal logics or binding of free variables.

3.4 Deployment

In this section, we present different aspect deployment methods and their application. We first illustrate a simple global deployment mechanism, which we will use for our debugging application (see Sec. 4.2). Then we define a deployment of aspects on object references and deployment in the scope of a block of code, which we will apply in our testing application (see Sec. 4.1).

Global Deployment Using the global deployment method, all deployed aspects have a global scope. Figure 4 shows the language interface and sample deployment. The aspect language `Global` subclasses `AspectLanguage`, which manages the registration of all language components. At (1), we initialize a global aspect environment that stores a list of deployed aspects (see Fig. 1 (6)). The methods

```
    class Global < AspectLanguage
     def initialize
(1)   @context = AspectEnvironment.new
     end
(2)  def processor; CustomProcessor end
     def context; @context; end
     def weaving jp
      context.iterate_aspects{ |a|
       a.each_advice(jp){ |pc, ad|
        if pc.evaluate(jp)
(3)      yield lambda{ad.evaluate(jp)}
        end }}
     end
    end
```

```
(4) def deploy aspect
     LANG.context.add aspect
    end
    def undeploy aspect
     LANG.context.remove aspect
    end
    # Example code:
    def get_sign x
     if x>0; ">0" else "<=0" end; end
    def sign x; puts get_sign(x); end
(5) deploy tracing
    sign(5)
    undeploy tracing
```

Fig. 4. Global deployment of the tracing aspect from Fig.3

at (2) define the language semantics. They are called during language initialization and from the join point shadows. We reuse the join point processor defined in Fig. 2. The `weaving` is performed at every join point shadow, it takes a meta join point and an anonymous block as arguments. The block guides advice application and is called with `yield` for each matching advice at (3). This abstracts as a co-routine over the actual weaving loop of both companion and interception types. Extensions of the weaving loop could comprise, e.g., advice precedence or optimizations that omit aspects at particular join points.

The deployment methods are defined at (4), and manipulate the global aspect environment (the aspect language is accessible via the global constant `LANG`). We deploy the tracing aspect from Fig.3 at (5) on a simple example. The application of the `sign` method causes tracing of `sign` and `get_sign` and of the evaluation of the condition in `get_sign`.

Per Object Deployment The first column of Fig. 5 shows deployment in the scope of particular objects. Instead of a global aspect environment, we maintain an environment per object. The objects are augmented using a central object manager (see Fig. 1 (10)), initialized at (1). It can be accessed using the `objects` method of the language. At (2), the weaving process iterates over the object stored in the target field of the join point, which is, e.g., the receiver object at method calls or the `self` reference at an if-condition join point. The deployment method at (3) creates a new aspect environment and adds it to the corresponding object. Applying the example at (4) using the tracing aspect from Fig. 3 will trace the method call `to_s` of `date`, but no methods from other objects.

Per Block Deployment The second column of Fig. 5 introduces deployment in the scope of a block of code. Instead of one global environment, an environment stack is initialized at (5) and maintained with the callback methods at (6). The deployment method at (7) takes an aspect, a (yet unused) scoping strategy and

```
    class PerObject < AspectLanguage
     def initialize
(1)   @objects = ObjMan.new(:env)
     end
     def weaving jp
(2)   if objects.augmented?(jp.target)
       env = objects.get_data(jp.target)
       env.iterate_aspects{ |a|
         ... }
      end
     end
    end

    def deploy_on object, a
(3)  env = AspectEnvironment.new
     env.add a
     LANG.objects.augment(object)
     LANG.objects.set_data(object, env)
    end

    # Example:
(4) date = Date.new(15, 3)
    deploy_on date, tracing
    puts date.to_s
```

```
    class PerBlock < AspectLanguage
     def initialize
(5)   @context = [AspectEnvironment.new]
     end
(6)  def context; @context.last; end
     def prop_up env
      @context.push env
     end
     def prop_down
      @context.pop
     end
    end

(7) def deploy a, s=nil
     env = LANG.context.clone
     LANG.prop_up(env)
     LANG.context.add a, s
     yield
     LANG.context.remove a, s
     LANG.prop_down
    end

    # Example:
(8) deploy tracing do sign(5) end
```

Fig. 5. Per object and per block deployment of the tracing aspect

an anonymous block as arguments. It maintains the aspect environment stack around the call to the block (`yield`), so that the deployed aspect is active in the dynamic extent of this block. The example at (8) will apply tracing to everything in the control flow of the method call `sign(5)`.

3.5 Scoping

Tanter generalized the scope of aspects with a set of propagation or scoping functions [21]. An aspect can, for example, be propagated over the call stack at particular join points or into object references to enable a so called *pervasive* scoping. Our design facilitates a lightweight integration of some or all of these scoping mechanisms as we will show in the following. A scoping strategy object stores the scoping functions and is associated with each aspect (shown in Fig. 5 at (7) as parameter `s`). It can be accessed at various points during weaving and propagation. Scoping functions have the same semantics as pointcuts in our language (see Sec 3.3) and can be extended in the same way. The propagation of aspects is performed by propagation shadows in the code. Those are inserted like join point shadows by the AST transformation. A propagation shadow embeds a piece of code like an interceptor shadow and inserts a propagation expression before and after the code. Possible expressions are `filter(x)`, where the current aspect environment is filtered by function `x`, `inject(x, f)`, where the aspect environment is filtered by `x` and stored in the actual join point using field `f`,

```
    class AroundMExecProp
        < PropagationShadowType
      def self.reification ast_context
        {:name => ast_context[:name]}
      end
      def self.meta ...
(1)   def self.prop_up
        filter(:c)
      end
    end
```

```
    class CustomProcessor ...
     def process_defn(exp) ...
      return s(:defn, name, args,
(2)         AroundMExecProp.sexp(body))
     end
    end
    # Example:
    s = Strategy.new {
(3)  {:c => pc{jp.name != :get_sign}}}
    deploy tracing, s do sign(5) end
```

Fig. 6. Scoping strategy over the call stack

and `extract(f)`, where the aspect environment is restored from the actual join point using field `f`.

Call Stack Scope Figure 6 defines the propagation of aspects over the call stack. The singleton class for propagation shadow creation (see Fig. 1 (3)) also has an associated meta join point for reification. We reuse the processor from Fig. 2 and the language `PerBlock` defined in Fig. 5, but augment method bodies with the propagation expression at (2). The propagation expression automatically calls the context propagation of the language (which we defined in Fig. 5 at (6)), so that the environment stack is extended in the context of the embedded code. Before the embedded code is executed, the propagation at (1) applies the filter method, which uses scoping function `c` of the scoping strategy associated with each aspect in the environment. The example at (3) defines a strategy that associates with function `c` a pointcut that prevents propagating the tracing beyond the dynamic extent of the `get_sign` method.

Delayed Evaluation Scope In object-oriented languages, an object created in the scope of a dynamic aspect can escape this scope through a reference (which would be a flaw for a security aspect [23]). The propagation of aspects into references allows a richer and more pervasive scoping [21]. We reuse the language defined in Fig. 6, augmenting object creation sites with a propagation shadow that applies `inject(d, result)` after the embedded code, which will store the actual environment in the result (the created object) filtered by `d`. At method executions, we use propagation `extract(target)` to restore the aspect environment from the receiver. Now we can apply the language in the following example:

```
s = Strategy.new do {:d => True} end
deploy tracing, s do; date = Date.new 17,4; end
puts date.to_s
```

We associate the strategy function `d` with a constant pointcut that always returns `true`, which propagates the tracing aspect into the escaping reference `date`.

```
class TestLangProcessor ...
 def process_if(e)
   # From Fig. 2
 end
 def process_defn(exp)
   # From Fig. 6
 end
end

# Example:
def create_display color, refresh
 log "Create display"
 if color
   # Creates colored display...
   if refresh; # ...with refresh
   else; # ...without refresh
   end
 end
end
```

```
    # Variant without lazy choice:
(1) create_display(choice(true, false),
        choice(true, false))

    # Variant with lazy choice:
    lazy_choice = ExtAspect.new do
     around if_cond do
       case (result = proceed)
         when TVTrue then true
         when TVFalse then false
(2)      when TVUnkn then
             choice(true, false)
         else result
       end; end; end

    s = Strategy.new {
        {:c => !name(:log)}}}
(3) deploy lazy_choice, s do
     create_display(TVUnkn, TVUnkn)
    end
```

Fig. 7. Aspect for lazy choice in testing with non-determinism

4 Applications

In the following, we present two instantiations of our MAP. They are used for the development of two different analyses that build on distinguished AOP features.

4.1 Explorative Testing

In earlier work, we presented a test exploration tool that reduces the size of test cases by applying a non-deterministic choice operator [2]. Such an operator has also been used for the generation of complex test input [12]. The authors of [12] suggest the application of lazy choice to avoid a combinatorial explosion of possible executions. In the following, we instantiate the MAP for integrating lazy choice with an aspect that delays choices to the evaluation of conditions.

The language components for instantiating the MAP are shown in Fig 7. We implement a 3-valued boolean abstraction with `TVTrue`, `TVFalse` and `TVUnkn`. We will reuse the stack propagation from Fig. 6 and the deployment on blocks from Fig. 5. Due to the direct evaluation of `choice`, the execution of the example at (1) will lead to four different execution paths of which two are identical. At (2), we define a lazy choice aspect that resolves 3-valued logic abstractions at the evaluation of conditions. Like that, the choice is delayed to the point were the *unknown* value flows into a condition. In the scope of the aspect, the test at (3) will yield only the three distinguished executions. For the sake of brevity, we omit caching of made choices in this example. With the deployment strategy `s`, we optimize the aspect's scope by avoiding propagation into logging methods of the program. While we saved only one execution, lazy choice becomes particularly important when generating more complex test input.

```
class AtStmt
    < CompanionShadowType
 def self.reification ast_context
  result = {:modifier => :before,
    :stype => ast_context[:stype]}
  if ast_context[:var]
   result[:var] =
     ast_context[:var]
  end
  return result
 end
 def self.meta ...
end

class CustomProcessor ...
 def process_if(exp)
  ast_context[:stype] = :if
  return s(:block,
     AtStmt.sexp, s(:if,...))
 end
 def process_lasgn(exp)
  ast_context[:stype] = :lasgn
  name = exp.shift
  ast_context[:var] = name
  return s(:block,
     AtStmt.sexp, s(:lasgn,...))
 end
end
```

```
    tracing = ExtAspect.new do
(1)  before stmt & stype(:lasgn) do
      puts "Assignment to: #{var}"
     end
    end

    debugging = ExtAspect.new do
     around execution do
      puts "Entering: #{name}"
      print "Step [i|o|t|u]:"
      com = gets.chomp
(2)   deploy(tracing) if com =~ /t/
      undeploy(tracing) if com =~ /u/
      if com =~ /o/
(3)    undeploy debugging
       begin
        proceed
       ensure
        deploy debugging
       end
      else
       proceed
      end
     end
(4)  before stmt ...
    end

    deploy debugging
    # Application code...
```

Fig. 8. Debugging language components and debugging aspect

4.2 Debugging

In the following, we develop a debugging-specific extension of our protocol and then build a prototype debugger based on it. The first column of Fig. 8 shows an excerpt of the join point processor. We define a statement-based join point model that introduces join point shadows, e.g., at assignments, conditions, loop headers and bodies, method calls, etc. We reuse some language components like method execution join points defined in Fig. 2, the global deployment mechanism from Fig. 4 and some simplifying pointcut and advice expressions. We minimize the reified context information for the sake of brevity, but it is straightforward to include more data, e.g., about the static nesting of each statement.

The second column of Fig. 8 shows a simple debugging aspect. A statement advice enables stepping per statement at (4). We reuse the tracing aspect from Fig 4 extended with statement-based tracing, e.g., variable assignment at (1). The tracing can be toggled on and off during debugging with a command at (2). When tracing is off, the tracing aspect is not deployed and does not produce additional runtime overhead. The around advice intercepts method executions to facilitate a *step over* or *step into* functionality at (3). If the user chooses *step over*, the debugging aspect undeploys itself in the dynamic extent of the method's `proceed`.

The example shows the instantiation of a fine-grained join point model that goes beyond general-purpose AOP. Together with the testing example it demonstrates the selection and combination of distinct AOP features. We expect that analyses in other domains like profiling or security can also be rapidly developed through different instantiations of our meta-aspect protocol.

5 Conclusion

We presented a meta-aspect protocol for tailoring analysis-specific aspect languages. Our discussion of dynamic analyses showed distinguished requirements on AOP in different analysis domains. We illustrated a broad spectrum of dynamic AOP features that can be selectively combined. Analysis-specific instantiations configure join point model, deployment and scoping, based on which the actual analysis aspects can be rapidly prototyped. We discussed two example analyses in the domains debugging and testing to validate the usefulness of the approach. We demonstrated how to separate the work of language and domain expert, which will greatly ease the rapid development and the maintenance of dynamic analyses in different domains.

Acknowledgments

The authors would like to thank Christian Hofer and all the anonymous reviewers for their insightful comments and suggestions.

References

1. Achenbach, M., Ostermann, K.: Growing a dynamic aspect language in Ruby. In: Proceedings of the 2010 AOSD Workshop on Domain-Specific Aspect Languages. ACM (2010)
2. Achenbach, M., Ostermann, K.: Testing object-oriented programs using dynamic aspects and non-determinism. In: Proceedings of the 1st ECOOP Workshop on Testing Object-Oriented Systems. ACM (2010)
3. Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding trace matching with free variables to AspectJ. In: OOPSLA '05. pp. 345–364. ACM (2005)
4. Aracic, I., Gasiunas, V., Mezini, M., Ostermann, K.: An overview of CaesarJ. Transactions on AOSD I, LNCS 3880, 135 – 173 (2006)
5. Binder, W., Villazón, A., Ansaloni, D., Moret, P.: @J: towards rapid development of dynamic analysis tools for the Java Virtual Machine. In: Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages. pp. 1–9. ACM (2009)
6. Bockisch, C., Haupt, M., Mezini, M., Ostermann, K.: Virtual machine support for dynamic join points. In: AOSD '04. pp. 83–92. ACM (2004)

7. Bodden, E., Havelund, K.: Racer: effective race detection using AspectJ. In: Proceedings of the 2008 International Symposium on Software Testing and Analysis. pp. 155–166. ACM (2008)
8. Bodden, E., Stolz, V.: Efficient temporal pointcuts through dynamic advice deployment. In: Open and Dynamic Aspect Languages Workshop (2006)
9. Copty, S., Ur, S.: Multi-threaded testing with AOP is easy, and it finds bugs! In: Parallel Processing, 11th International Euro-Par Conference. LNCS, vol. 3648, pp. 740–749. Springer (2005)
10. Dinkelaker, T., Mezini, M.: Dynamically linked domain-specific extensions for advice languages. In: Proceedings of the 2008 AOSD Workshop on Domain-Specific Aspect Languages. pp. 1–7. ACM (2008)
11. Dinkelaker, T., Mezini, M., Bockisch, C.: The art of the meta-aspect protocol. In: AOSD '09. pp. 51–62. ACM (2009)
12. Gligoric, M., Khurshid, S., Gvero, T., Kuncak, V., Jagannath, V., Marinov, D.: Test generation through programming in UDITA. In: ICSE '10. pp. 225–234. ACM (2010)
13. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: ECOOP '01. pp. 327–353. Springer (2001)
14. Kiczales, G., Rivieres, J.D., Bobrow, D.G.: The Art of the Metaobject Protocol. MIT Press, Cambridge, MA, USA (1991)
15. Maebe, J., Buytaert, D., Eeckhout, L., De Bosschere, K.: Javana: a system for building customized Java program analysis tools. In: OOPSLA '06. pp. 153–168. ACM (2006)
16. Nir-Buchbinder, Y., Ur, S.: ConTest listeners: a concurrency-oriented infrastructure for Java test and heal tools. In: Fourth International Workshop on Software Quality Assurance. pp. 9–16. ACM (2007)
17. Nusayr, A., Cook, J.: AOP for the domain of runtime monitoring: breaking out of the code-based model. In: Proceedings of the 4th Workshop on Domain-Specific Aspect Languages. pp. 7–10. ACM (2009)
18. Parse Tree and Ruby Parser. `http://parsetree.rubyforge.org/`
19. Rakesh, M.G.: A lightweight approach for program analysis and debugging. In: Proceedings of the 3rd India Software Engineering Conference. pp. 13–22. ACM (2010)
20. Ruby2Ruby. `http://seattlerb.rubyforge.org/ruby2ruby/`
21. Tanter, É.: Expressive scoping of dynamically-deployed aspects. In: AOSD '08. pp. 168–179. ACM (2008)
22. Tanter, É.: Beyond static and dynamic scope. In: Proceedings of the 5th ACM Dynamic Languages Symposium. pp. 3–14. ACM (2009)
23. Toledo, R., Leger, P., Tanter, É.: AspectScript: Expressive aspects for the Web. In: AOSD '10. ACM (2010)
24. Villazón, A., Binder, W., Ansaloni, D., Moret, P.: Advanced runtime adaptation for Java. In: GPCE '09. pp. 85–94. ACM (2009)
25. Villazón, A., Binder, W., Ansaloni, D., Moret, P.: HotWave: creating adaptive tools with dynamic aspect-oriented programming in Java. In: GPCE '09. pp. 95–98. ACM (2009)
26. Visser, W., Havelund, K., Brat, G., Park, S.: Model checking programs. In: Automated Software Engineering. pp. 3–11. IEEE (2000)