A Lightweight and Extensible AspectJ Implementation[1]

Rodolfo Toledo
(PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile, Santiago, Chile
rtoledo@dcc.uchile.cl)

Éric Tanter
(PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile, Santiago, Chile
etanter@dcc.uchile.cl)

Abstract: Extending AspectJ to experiment with new language features can be cumbersome, even with an extensible implementation. Often, a language designer only needs a rapid prototyping environment, but has to deal with a full compiler infrastructure, and must address low-level implementation issues. This work completes a lightweight extensible implementation of AspectJ with a declarative assimilation layer based on Stratego. This layer brings together an extensible syntax definition of AspectJ and the core semantics provided by the Reflex AOP kernel. Using this implementation, language extensions are defined using declarative high-level constructs, significantly reducing the cost of the extension process.

Key Words: Aspect-oriented programming, AspectJ, extensible implementation
Category: D.1.m, D.3.4

1 Introduction and Motivation

A concern is said to *crosscut* an application if its code is spread in several modules. Examples of this kind of concerns are: logging, synchronization and error-handling, among others. The consequence of the existence of these concerns is that they can not be implemented in a modular way using traditional paradigms. Aspect-Oriented Programming (AOP) was proposed as a new paradigm to solve these modularization problems [Elrad et al. 2001]. AOP introduces the notion of an aspect: a modular implementation of a crosscutting concern.

AspectJ [AspectJ Website 2002] is one of the most popular languages for AOP in Java. It allows the definition of aspects by means of *pointcuts* and *advices* (among other features). An advice describes the action to take when a *join point* (an execution point) is matched by a pointcut; a pointcut is a composable descriptor that matches a set of join points based on lexical or

[1] This work is partially funded by FONDECYT Project 11060493.

dynamic conditions. The *weaving* process connects the application code to the aspects definitions.

17 kinds of pointcuts designators for selecting execution points can be found in AspectJ. Some of them are *static* designators as their join points can be determined statically given their kind (method invocation, field read access, etc.) and their lexical pattern. Examples are `call` and `get` that select join points of the aforementioned kinds. Other class of pointcuts are *dynamic* designators, which impose dynamic restrictions over a join point. For example, `if` restricts the match to obey an arbitrary condition. Finally, some other pointcuts are *context-sensitive* because they can expose context information present at the join point or impose a type restriction over that context. For example, `this` can be used either to expose the current object at the join point or to impose a type restriction on it; similarly with `args` and `target`.

Extending AspectJ. Despite the number of AspectJ pointcuts, the exploration of new ones is a research issue [Avgustinov et al. 2006, Gybels and Brichau 2003, Harbulot and Gurd 2006, Ostermann et al. 2005, Tanter et al. 2006], mostly motivated by the quest for increased expressiveness. Unsurprisingly, many AspectJ extensions are focused on pointcuts. Among these extensions we can find new pointcuts like `cast` and `global` in the EAJ extension [Avgustinov et al. 2006]. The `cast` pointcut matches *type casts* events; and the `global` pointcut allows the factorization of a common pointcut expression in one global declaration.

Other extensions to the pointcut language of AspectJ are proposed for context-aware aspects [Tanter et al. 2006]. Two general-purpose pointcuts descriptors are presented: `inContext` imposes a restriction enforcing a certain context to be currently active at the join point; `createdInCtx` checks that the current object at the join point was created when a certain context was active. These pointcuts are similar to an `if` pointcut, except that they can be used to expose external context values to the advice. [Tanter et al. 2006] also motivates the need for being able to easily define *domain-specific* pointcuts, especially tailored for an application or domain.

Even though some of these context-aware pointcuts (and maybe other extensions to the pointcut language) can be simulated using standard AspectJ, the fundamental difference resides in expressiveness. For example, using plain AspectJ, one aspect would be necessary just to record the context in which an object is created. This aspect must also provide an interface to access that information, necessary to determine if a pointcut restricted to objects created in a certain context matches or not.

Researchers wanting to experiment with AspectJ extensions face the problem that extending the standard AspectJ compiler (`ajc`) is hard and implementing a customized version of it from scratch is not an option. This is the motivation behind `abc` [Avgustinov et al. 2006], an extensible compiler for AspectJ.

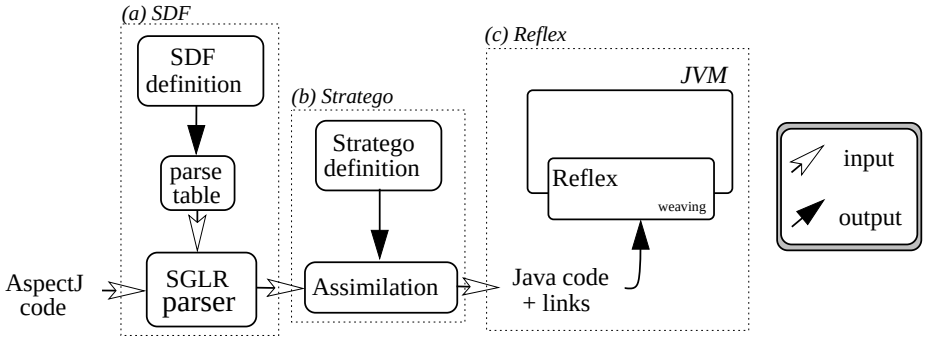

Figure 1: ReflexBorg architecture.

Contributions. Our stake on this issue is that some simple extensions do not require all the machinery of a full-fledged compiler like `abc`. When initially experimenting with such extensions, a lightweight extensible implementation of AspectJ suffices, allowing rapid prototyping. At this stage, performance considerations are not necessarily an issue. When the extension matures and optimizations are considered, then turning to a full compiler infrastructure makes sense.

The contribution of this work is to provide an extensible and lightweight AspectJ implementation over a declarative intermediate language (Section 3) based on the ReflexBorg approach (Section 2). We demonstrate its extensibility by actually extending it to support the `cast` and `global` pointcuts, and context-aware aspects (Section 4). We validate our approach by comparing the implementation of these extensions in ReflexBorg and `abc` (Section 5). Section 6 concludes.

2 ReflexBorg Approach

The ReflexBorg approach is a method for implementing aspect-oriented extensions of Java, including both their syntax and semantics. This work reports on completing an AspectJ implementation using ReflexBorg.

ReflexBorg consists of three layers (Figure 1): syntax definition of the language (or its extensions) are expressed in SDF (Sect. 2.1); ultimately, the semantics of the language are realized in Reflex (Section 2.3), which takes care of aspect weaving. In between, an assimilation layer defined in Stratego (Section 2.2) transforms abstract terms of the aspect language into Java code instantiating Reflex elements.

2.1 SDF

SDF (Syntax Definition Formalism) is an extensible and modular language for defining syntax [Visser 1997]. Definitions (derivations, lexical restrictions, terminals, keywords, etc.) are done in modules that can be extended and reused. An SDF grammar is processed by the SGLR parser generator to generate a parse table. With this parse table and the provided scannerless generalized-LR (SGLR) parser, abstract syntax is obtained (Figure 1a). A major interest of SGLR parsing technique is that it can parse any context-free grammar, not only the LL or LALR subclasses; this is interesting because context-free grammars are closed under composition. For these reasons, GLR is specially well-suited compared to other technologies for parsing complex languages that are a composition of substantially different languages. AspectJ is made up of the Java language, the pointcut language and the advice language; it can be defined in SDF in a declarative, formal and extensible way [Bravenboer et al. 2006a].

2.2 Stratego/XT

Stratego and XT [Visser 2004] represent a powerful machinery for program transformation. Stratego is a declarative language for transforming trees through the application of rewrite *rules*, composed by means of rewriting *strategies* for modular transformation and fine-grained control over their application. Rewrite rules can use the concrete syntax of the host language in their definitions. Stratego also provides *dynamic* rewrite rules, for context-sensitive program transformation [Bravenboer et al. 2006b]. Dynamic rewrite rules can be progressively defined at runtime (that is, during rewriting itself) and can therefore use dynamic context information to drive further rewriting.

Finally, XT is a toolset which offers a collection of extensible and reusable transformation tools such as the SGLR parser used in conjunction with SDF. The transformation process of AspectJ into Reflex (Figure 1b) will hereafter be called *assimilation* (after [Bravenboer and Visser 2004]).

2.3 Reflex

Reflex is a Java implementation of a versatile kernel for aspect-oriented programming using bytecode transformation [Tanter and Noyé 2005]. The role of Reflex in the ReflexBorg approach is to act as a weaver, connecting the application code and aspects (Figure 1c). As a general-purpose kernel for reflective and metaprogramming, Reflex provides basic building blocks for implementing aspects in Java, whose expressiveness covers the range of AspectJ features [Rodríguez et al. 2004]. As an AOP framework, Reflex has been used to implement several extensions, including the core semantics of context-aware aspects [Tanter et al. 2006].

Links. The central abstraction to drive weaving in Reflex is that of explicit links binding a set of program points (a hookset) to a metaobject. A link is characterized by a number of attributes, e.g., the control at which metaobjects act (before, after, around), and a dynamically-evaluated activation condition. A link therefore corresponds to a single pointcut/advice pair.

Hooksets. Program points are described as occurrences of *operation classes*, such as `MsgSend`, and `FieldRead`. A hookset uses an *operation selector* to select points of interest, occurring within classes specified by a *class selector*. When a link is deployed, hooks are inserted appropriately to provoke reification at runtime, following the protocol specified for each link delegating the control to the specified metaobject.

Metaobjects. A metaobject implements the action associated to an aspect. In Reflex, it can actually be any standard Java object.

2.4 Contribution

Figure 2 clarifies the context in which the present work is developed. The SDF definition of AspectJ, altogether with several extensions including EAJ and context-aware aspects (CAA) was described in [Bravenboer et al. 2006a]. Reflex has been used to implement AspectJ in [Rodríguez et al. 2004]; context-aware aspects were implemented as a Reflex framework in [Tanter et al. 2006]; and the building blocks for implementing EAJ are provided by [Tanter and Noyé 2005]. The contribution of this work is to complete the extensible implementation of AspectJ by providing the missing middle layer: the Stratego assimilation of AspectJ abstract syntax into Reflex definitions (links), and showing its extensibility by considering both EAJ and context-aware aspects.

3 Defining AspectJ in ReflexBorg

Consider the following AspectJ aspect that prints a message to standard input at the beginning of the `main` method:

```
aspect HelloWorld {
  void before(String[] argsv) :
      call(void main(String[]) && args(argsv) {
    System.out.println("Hello World " + argsv[0] + "!");
} }
```

The code above uses `call` to select the `main` method and `args` to expose the parameter `argsv`. Listing 1 shows the result of its assimilation: advices are translated into Java methods and static pointcuts into hooksets; the link binding these elements is configured depending on what is specified in the advice and

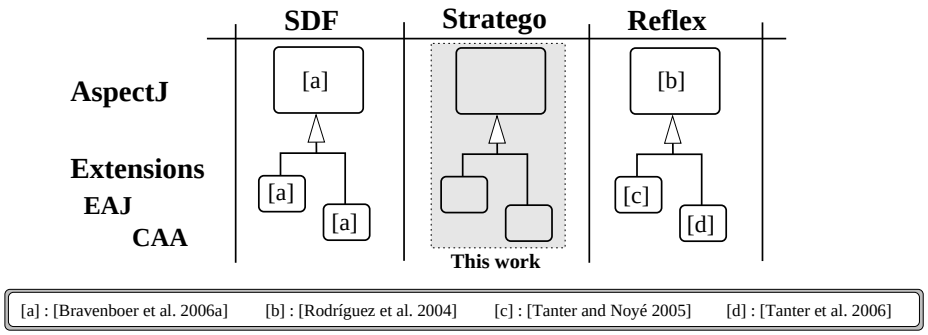

Figure 2: Elements of the ReflexBorg approach for an extensible AspectJ implementation.

Listing 1 HelloWorld aspect assimilation into Reflex

```
class HelloWorld {
  void adv_1(String[] argsv){
    System.out.println("Hello World " + argsv[0] + "!");
  }
  public void initReflex(){
    Link link = API.links().createLink(new HelloWorld());
    link.addHookset(new Hookset(MsgSend.class, new AllCS(),
                                new AJOS("void main(String[]")));
    link.setControl(Control.BEFORE);
    link.setCall("HelloWorld", "adv_1", new IdxParam(1));
} }
```

the pointcuts, such as the exposed context information (`argsv`) and the kind of the advice (`before` in this case).

Previous experience tells us that taking a direct approach to assimilate AspectJ into Reflex definitions results in overly complex Stratego rules. The reasons for this complexity are several. First, the inherent complexity of AspectJ where, for example, arguments passed to the advice are not specified together but are spread in several pointcut designators. Second, the mismatch between AspectJ and Reflex constructs: most pointcuts are assimilated into hooksets, advices into methods, but the communication protocol between the base code and the advice has not direct equivalence. And last but not least, the complexity associated to the generation of valid Java code.

Listing 1 perfectly exemplifies the complexity involved in the direct assimilation of AspectJ to Reflex. First, the value of `argsv` is determined by the `args` pointcut, but needs to be placed along other information to conform the `setCall` invocation; this gathering of information introduces coupling in the

assimilation of different AspectJ constructs. Second, the information necessary to construct the `setCall` method must be extracted from several AspectJ constructs, highlighting the mismatch between AspectJ and Reflex: the metaobject type (`HelloWorld`) comes from the name of the aspect, the method name (`adv_1`) from the name given to the advice and the argument from the `args` pointcut. Finally, the order in which the statements are placed in the final code is important: the link must be declared before it is actually used.

To tackle all the aforementioned issues, we first introduce a intermediate declarative language for the assimilation of AspectJ into Reflex: DKLang (Section 3.1). In addition to DKLang, we define a stratified assimilation process (Section 3.2). The objective of this stratification is to reduce even more the coupling in the assimilation of different AspectJ constructs.

3.1 Declarative Kernel Language

DKLang is a declarative interface to Reflex. The main abstraction, as in Reflex, are links. Each link has a number of properties corresponding to the ones we can find in Reflex. Some properties can take their values from a finite set of alternatives, like the `control` property; others can take an arbitrary Java expression, like the `hooksets` property. DKLang includes an assimilation layer in charge of producing the equivalent Reflex definitions. This layer ensures that syntactically-correct Java code is finally generated.

One important feature of DKLang is that its assimilation into Reflex definitions is immune to the order of the declarations. This makes irrelevant the order in which the original constructs are assimilated. This feature is fundamental for the decoupling of the AspectJ assimilation process. This is in contrast with the first attempt at defining a language for Reflex [Tanter 2006b].

DKLang is designed to decouple as much as possible the assimilation of different constructs of AspectJ. Despite the fact that DKLang was originally designed for AspectJ, we have used it also in the assimilation of KALA, a language for advanced transaction management [Fabry and D'Hondt 2006] and in the assimilation of a extension of KALA with the pointcut language of AspectJ.

The basic syntax of DKLang can be summarized as:

```
linkID "." propertyName ["="|"+="] propertyValue
```

`linkID` can be any normal Java identifier and serves to discriminate among different links. `propertyName` corresponds to the name of the property being defined and `propertyValue` to the actual value being set. There are two possible operators depending on the property being set: `=` and `+=`, for single-valued and multiple-valued properties respectively.

Let us revisit the example of Listing 1. We can observe the complete decoupling in the assimilation of different AspectJ constructs if we use DKLang

(Listing 2). The reason is that the assimilation of each AspectJ construct produces one (or more) DKLang declarations without having to interact with other constructs. For example, the components of the call to the metaobject (type, method name and parameters[2]) do not need to be combined explicitly: only individual declarations are necessary as the combination and generation of valid Reflex definitions are handled automatically by the DKLang assimilation layer.

Listing 2 HelloWorld aspect expressed in DKLang

```
link.mo = new HelloWorld()
link.hooksets += new Hookset(MsgSend.class, new AllCS(),
                             new AJOS("void main(String[]")))
link.control = before
link.call.type = "HelloWorld"
link.call.method = "adv_1"
link.call.params[1] = $params[1]
```

In addition to properties, DKLang also supports per-link variables. This allows each link to define a local scope for auxiliary variables used during the assimilation. These variables allow the assimilation of different AspectJ constructs to cooperate in the production of a link. A variable expression can be used anywhere an arbitrary Java expression can, *e.g.* in `hooksets` and `call.method` properties. Since variables are link-local, no name clashes occur between links, ensuring an hygienic assimilation. The examples we give in the remainder of this paper make us of link variables.

It is important to notice the difference between link *properties* and link *variables*. Link properties are predefined and correspond to the attributes of links in Reflex. Link variables can be defined freely and their existence only depends on the particular assimilation of a construct. Link variables also differ in their syntax: a variable `x` in link `l` is referred to as `l[x]`.

3.2 Assimilation Stages

The overall structure of the assimilation process is described in Figure 3. There are two main stages called *general rewriting* and *top-down assimilation*.

3.2.1 General Rewriting Stage

The general rewriting stage provides an entry point for extensions that want to operate on the abstract syntax tree (AST) before it starts being assimilated. For instance, we make use of a general rewriting rule that inlines all the named pointcuts, leaving the AST normalized (this prevents each extension from providing

[2] We use `$params[1]` to access the value of the first parameter of the call to `main`

its own pointcut name binding mechanism). This stage can be used for other purposes, such as static analysis, although this dimension has not been explored yet. Extensions must declare a `GeneralRewrite` rule matching the entire AST. The implementation of this stage is as follows:

```
MainGeneralRewrite :
    ast -> new-ast
  where
    new-ast := <repeat(GeneralRewrite)> ast
```

First, the `MainGeneralRewrite` rule matches the original `ast`. Then, it applies the `GeneralRewrite` until it fails. When this failure occurs, the result of the successive repetition of `GeneralRewrite` is returned through the `new-ast` variable.

Extension writers ought to implement the `GeneralRewrite` rule in a way so it only succeeds once. If several implementations are provided, they are applied in an unspecified order. This does not represent a limitation as extension writers can still compose rules using Stratego [Visser 2004] (for example, sequences can be specified by `s1; s2`).

Example of use: global Pointcut Assimilation.

The purpose of the `global` pointcut is to factorize a set of pointcut restrictions in one declaration. It is usually used to specify some global restriction on already-defined aspects or to adapt a generic aspect to different needs.

Consider the following example that protects the implementation details of the `HiddenImplementation` class from all aspects whose name do not start with `Privileged` [Avgustinov et al. 2006]:

```
class HiddenImplementation {
  global : !Privileged* : !within(HiddenImplementation);
  ...
}
```

The abstract syntax in SDF of the `global` pointcut declaring it as a new type of pointcut declaration is [Bravenboer et al. 2006a]:

```
"global" ":" ClassNamePattern ":" PointcutExpr ";" -> PointcutDec
```

SDF declarations are expressed in inverse BNF notation. In this case, a `PointcutDec` must start with the `global` keyword followed by a colon, after that, a `ClassNamePattern`, another colon and a `PointcutExpr`. The whole pointcut declaration ends with a semicolon.

To assimilate this pointcut we use a general rewriting rule (step ① Figure 3). Following is the main part of the general rewriting rule:

```
GeneralRewrite :
 CompilationUnit(pkg,imports,decls) ->                    1
   CompilationUnit(pkg,imports,new-decls)                 2
```

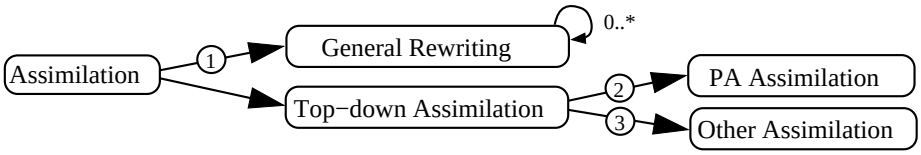

Figure 3: Assimilation process.

```
where
  global-decls := <collect-all(?Global(_,_))> decls ;          3
  i-decls      := <map(inline-globals(|global-decls))> decls ; 4
  new-decls    := <remove-all-rec(?Global(_,_)|)> i-decls      5
```

This rule accomplishes the effect of expanding the factorization expressed in the `global` declaration: line (1) matches the `CompilationUnit` node, which represents the AST of the file being assimilated; and then replaces its declarations with new ones obtained after the assimilation (line (2)). Line (3) collects all the global declarations, line (4) inlines them and finally, line (5) recursively removes the global declarations from the AST.

The `inline-globals` rule used above (line 4) is based in this rule:

```
inline-global : pointcut-expr -> |[ (pointcut-expr) && global-decl ]|
```

Here a new conjunction is constructed between the original pointcut expression and the global declaration. The `|[...]|` syntax is provided by Stratego to allow the use of concrete syntax in the assimilation rules. In our case, we use this feature to assimilate using AspectJ syntax (as in the code above) or DKLang syntax (later in this paper).

3.2.2 Top-down Assimilation Stage

This stage assimilates AspectJ constructs into Reflex links. The process consists in traversing the AST top-down, assimilating one sub-tree at a time, in two substages (Figure 3): one for assimilating pointcuts and advices (PA), and another for other AspectJ features, such as inter-type declarations (not discussed here).

The assimilation of a pointcut/advice pair is done using DKLang concrete syntax. The result of this assimilation is a new Reflex link. We focus here on the assimilation of pointcuts, since they are the most frequent locus of extension.

The first step in this stage is, for each advice, to expand the pointcut expression into a disjunctive normal form. The objective of this expansion is to process one conjunction of pointcut designators at a time. Then, for each pointcut designator, an `Assim-Pc` rule matching that particular kind of pointcut is invoked.

This rule is in charge of assimilating the AspectJ pointcut into DKLang declarations. To accomplish this assimilation, the rule is provided with two parameters: the advice formal parameters and a list of other pointcuts in the same conjunction. Due to the functional nature of Stratego, these parameters can only be read and not overwritten. This avoids side effects resulting from the assimilation of a particular pointcut, fostering the decoupling.

When each pointcut in the conjunction has been assimilated into DKLang declarations, no further configuration is needed because the assimilation layer of DKLang takes care of obtaining the final link. Single-valued properties are combined if necessary (*e.g.*`setCall` method arguments). Multi-valued properties are also combined according to their particular semantics (*e.g.*values of the `hookset` property are combined using a special-purpose Reflex hookset that computes their intersection).

Example of use: `cast` pointcut assimilation.

The `cast` pointcut is intended for intercepting occurrences of *type cast* events matching a lexical pattern. Consider the following advice that prints a warning whenever a downcast with loss of precision is performed [Avgustinov et al. 2006]:

```
before(int i): cast(short) && args(i) &&
               if(i < Short.MIN_VALUE || i > Short.MAX_VALUE) {
  System.err.println("Warning: casting " + i + " to a short.");
}
```

The abstract syntax of the `cast` pointcut in SDF declaring it as a new kind of pointcut expression is [Bravenboer et al. 2006a]:

```
"cast" "(" TypePattern ")" -> PointcutExpr
```

The `cast` pointcut falls into the static pointcuts category because it relies only on a lexical pattern. Since Reflex already supports altering cast occurrences (by means of the `Cast` operation class), the assimilation of this pointcut is straightforward (Listing 3): it augments the `hooksets` property of the current link with the appropriate operation class and selectors (lines (6)-(7)). The current link identifier is obtained using the `this-link` rule (line (8)), provided by the assimilation layer of DKLang. Due to space limits, we do not describe type pattern assimilation here.

4 Context-Aware Aspects Extension

In the previous section we introduced DKLang and the stratified assimilation process. We exemplified their use through the assimilation of the `global` and `cast` pointcuts (both part of Extended AspectJ [Avgustinov et al. 2006]). In this section we show how to define the context-aware aspects extensions as in [Tanter et al. 2006].

Listing 3 cast Pointcut Assimilation.

```
Assim-Pc(|formalParams,context) :
  Cast(type-pattern) -> |[                                          6
     ~link-id .hooksets += new Hookset(Cast.class, ~cs, ~os)        7
     ]|
  where
    link-id := this-link ;                                          8
    cs      := <assim-typepattern-cs> type-pattern ;
    os      := <assim-typepattern-os> type-pattern
```

4.1 Context-Aware Aspects

Context awareness is the ability to reason about the surrounding context of an application. This awareness could mean to know in which context the application is at some point in time, or in which context it was when some past event occurred. Context-aware aspects [Tanter et al. 2006] introduce two new general-purpose pointcuts: `inContext`, a restriction over the currently-active context; and `createdInCtx`, a restriction over the context in which an object was created.[3] An example of a context-aware aspect is:

```
aspect Discount {
  pointcut amount(double rate):
    execution(double ShoppingCart.getAmount()) &&
    inContext(StockOverloadCtx[.80]) && inContext(PromotionCtx(rate));

  double around(double rate): amount(rate) {
    return proceed() * (1 - rate);
} }
```

The `Discount` aspect uses an advice to apply a discount of the specified `rate` to the total price of the shopping cart. The advice replaces the execution of the `getAmount` method of a `ShoppingCart` *only if both* the `StockOverloadCtx` and the `PromotionCxt` contexts are active. The `StockOverloadCtx` is parametrized to be active only when an 80% threshold is reached. The `PromotionCtx` determines the actual discount rate, which is exposed to the advice following the AspectJ context information exposure mechanism.

4.2 Assimilation of Context-Aware Aspects

In [Tanter et al. 2006], the authors develop an open framework for context-aware aspects implemented over Reflex as a class library, but do not implement the actual extension. We therefore choose to assimilate context-aware aspects into this framework.

Below is the result of assimilating the `inContext(PromotionCtx(rate))` pointcut of the `Discount` aspect.

[3] Due to space limitations we focus on the implementations of these general-purpose pointcuts, and not on the domain-specific ones introduced in [Tanter et al. 2006].

```
Context context_1 = new PromotionCtx();
CtxActive activation_1 = new InContext(context_1);
link.addActivation(activation_1);
link.setCall("Discount", "adv_1", activation_1.getCtxParam("rate") );
```

First, a `Context` object is created. This object represent the current promotion context. Then, a Reflex activation in created to restrict the execution of the metaobject (*i.e.*advice) only when the application is in the promotion context. Finally, the framed expression corresponds to the `rate` parameter, which is passed to the metaobject.

Listing 4 presents the assimilation rules for the `inContext` pointcut (rules for `createdInCtx` are very similar). The `Assim-Pc` rule matches an `InContext` pointcut node (9). Line (10) creates the object representing the context with its corresponding parameters and then, the object representing the dynamic restriction (11). Then, the current link is configured to obey that restriction (12). Finally, the parameters to be passed to the advice are determined by the application of the `assim-adv-param` rule (14). This rule uses the position in which a parameter appears in the signature of the advice (15) and the activation object (16) to set the `call.params` property of the link.

Listing 4 inContext Pointcut Assimilation.

```
Assim-Pc(|formalParams,context) :
  InContext(ActualCtx(TypeName(x),params,ctx)) -> |[                    9
     ~this-link [context] = new ~id:x (~*caa-params);                   10
     ~this-link [activation] = new InContext(~this-link [context]);     11
     ~this-link .activation = ~this-link [activation]                   12
     ~*link-params
  ]|
  where
    caa-params := <assimilate-params> params ;                          13
    ~link-params := <map(assim-adv-param(formalParams)> ctx             14

assim-adv-param(|formalParams) :
  x@Id(i) -> |[
    ~this-link .call.params[~idx] =                                     15
                ~this-link [activation].getCtxParam(~i)                 16
  ]|
  where
    idx := <get-index>(x, formalParam)
```

5 Evaluation

In this section we report on a first experimental study of the lightweight property of the ReflexBorg approach. We compare the implementation of the `cast`, `global` and `inContext` pointcuts in both ReflexBorg and `abc`. Extensibility is very hard

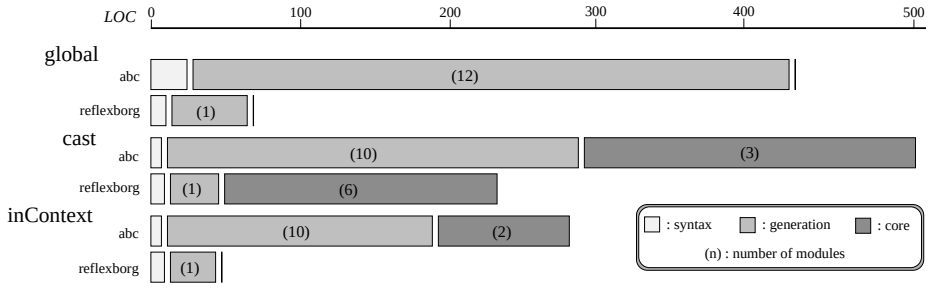

Figure 4: Size of new pointcut definitions in `abc` and ReflexBorg.

to measure, so our study adopts objectives (and discussable) metrics like number of lines of code and number of modules to define. These numbers are taken *in isolation*: we consider, for each new pointcut, that the starting point is the core system, naked. That is, we do not consider ad hoc reuse strategies between extensions. Results are given on Figure 4.

For each pointcut, we indicate the number of lines of code to define the extension. We differentiate between three steps: *a)* the syntax definition (SDF in ReflexBorg, PPG in `abc`), *b)* the definition of the generation phase (Stratego in ReflexBorg, AST definitions and analysis in `abc`), and *c)* the extensions to the core framework (*e.g.* a Reflex library). For each step, we indicate the number of modules (*e.g.* Java classes) to define.

For the syntax, because the extensions are fairly standard, both approaches (SDF for ReflexBorg and PPG for `abc`) give approximately the same results. However, the declarative nature of SDF is definitely a plus in terms of expressiveness. Also, previous work has shown the advantages of SDF over PPG for more intricate cases of AspectJ-like syntax [Bravenboer et al. 2006a]. For the two other steps, it is clear that `abc` requires a lot more code to be written than ReflexBorg. On average we can observe that using ReflexBorg results in having to write from 2 to 5 times less code than with `abc`.

The difference is mainly due to the following: *(i)* `abc` is a full compiler infrastructure, so it is necessary to deal with many optimization-related elements and code generation with low-level structures, while Reflex provides a relatively simple reflective model with few higher-level abstractions; *(ii)* the generation phase in ReflexBorg is doubly declarative: the language used to specify transformation, Stratego, is declarative, and the target language, DKLang, as well; in contrast, in `abc` this step has to be specified in plain Java, dealing with intricate inheritance and delegation patterns. Of course, these results serve only as a preliminary validation of the lightweight nature of ReflexBorg extensions. They

should be taken with care and more work is needed to be able to generalize them to any kind of AspectJ extensions.

6 Discussions and Conclusions

Limitations. At the time of this writing, our assimilation layer does not support certain features of AspectJ: pointcuts such as advice execution and exception handlers; `declare error/warning`; reflective join point information; and aspect precedence. Most of these features are direct to support since Reflex provides the necessary mechanisms, in particular with respect to aspect composition [Tanter and Noyé 2005] and static weaving [Tanter 2006a]. Also, a full AspectJ implementation would need a type checker. At present the assimilation process assumes type-correct input code, otherwise incorrect code can be generated. The general rewriting stage can be used to perform this kind of static analysis, among others.

Related Work. In [Rodríguez et al. 2004], an implementation of AspectJ over Reflex is presented. Their work is based on the modification of the AspectJ compiler to produce, instead of bytecode, Reflex links. They show that the model of Reflex is expressive enough for supporting AspectJ, but the solution is not extensible at all. Indeed, their implementation is tied to one version of the AspectJ compiler. Since they do not consider the definition of the syntax, adding new extensions is barely feasible. In contrast, our solution is designed to be extensible at all levels (recall Figure 2).

Josh [Chiba and Nakagawa 2004] is an open compiler for an AspectJ-like language. In Josh, it is possible to extend the available set of pointcuts through the specification of static methods that can alter the application code using low-level transformations. In contrast, the ReflexBorg approach provides high-level declarative abstractions to implement language extensions.

As mentioned before, `abc` [Avgustinov et al. 2006] is a full-fledged AspectJ open compiler, designed for extensibility. The key issue to consider when choosing between `abc` and ReflexBorg is whether the full power of a complete compiler is needed, or if a rapid-prototyping approach is enough. For studying efficient implementations of aspect language features, in particular using advanced static analysis, `abc` is definitely the alternative of choice [Avgustinov et al. 2005].

Conclusions. To address the issue of rapid prototyping of AspectJ language extensions, we presented a lightweight and extensible implementation of AspectJ that relies on the ReflexBorg approach. The implementation is extensible at all levels: syntax (SDF), assimilation (Stratego) and implementation/weaving (Reflex). This work has focused on the assimilation layer of ReflexBorg for

AspectJ which is based on a declarative language and several stages for increased decoupling of different constructs. We have demonstrated its extensibility by showing various extensions and comparing them to their definitions in the AspectBench Compiler. Future work includes integrating more AspectJ features, and experimenting with more demanding AspectJ extensions in order to refine our extensibility comparisons.

Acknowledgments

We thank Martin Bravenboer, Eelco Visser, and the Stratego project members for their support in the development of ReflexBorg. Part of this work was developed during the visit of Rodolfo Toledo at Delft University, under the guidance of Martin Bravenboer.

References

[AspectJ Website 2002] AspectJ Website (2002). The AspectJ website. http://www.eclipse.org/aspectj.

[Avgustinov et al. 2005] Avgustinov, P., Christensen, A. S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., and Tibble, J. (2005). Optimising AspectJ. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI 2005)*, pages 117–128, Chicago, USA.

[Avgustinov et al. 2006] Avgustinov, P., Christensen, A. S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., and Tibble, J. (2006). abc: an extensible AspectJ compiler. In *Transactions on Aspect-Oriented Software Development*, volume 3880 of *Lecture Notes in Computer Science*, pages 293–334. Springer-Verlag.

[Bravenboer et al. 2006a] Bravenboer, M., Tanter, É., and Visser, E. (2006a). Declarative, formal, and extensible syntax definition for AspectJ – a case for scannerless generalized-LR parsing. In *Proceedings of the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2006)*, pages 209–228, Portland, Oregon, USA. ACM Press.

[Bravenboer et al. 2006b] Bravenboer, M., van Dam, A., Olmos, K., and Visser, E. (2006b). Program transformation with scoped dynamic rewrite rules. *Fundamenta Informaticae*, 69(1–2):123–178.

[Bravenboer and Visser 2004] Bravenboer, M. and Visser, E. (2004). Concrete syntax for objects. In *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2004)*, pages 365–383, Vancouver, British Columbia, Canada. ACM Press. ACM SIGPLAN Notices, 39(11).

[Chiba and Nakagawa 2004] Chiba, S. and Nakagawa, K. (2004). Josh: An open AspectJ-like language. In Lieberherr, K., editor, *Proceedings of the 3rd ACM International Conference on Aspect-Oriented Software Development (AOSD 2004)*, pages 102–111, Lancaster, UK. ACM Press.

[Elrad et al. 2001] Elrad, T., Filman, R. E., and Bader, A. (2001). Aspect-oriented programming. *Communications of the ACM*, 44(10).

[Fabry and D'Hondt 2006] Fabry, J. and D'Hondt, T. (2006). KALA: Kernel aspect language for advanced transactions. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 1615–1620.

[Gybels and Brichau 2003] Gybels, K. and Brichau, J. (2003). Arranging language features for more robust pattern-based crosscuts. In Akşit, M., editor, *Proceedings of the 2nd ACM International Conference on Aspect-Oriented Software Development (AOSD 2003)*, pages 60–69, Boston, MA, USA. ACM Press.

[Harbulot and Gurd 2006] Harbulot, B. and Gurd, J. (2006). A join point for loops in AspectJ. In *Proceedings of the 5th ACM International Conference on Aspect-Oriented Software Development (AOSD 2006)*, pages 63–74, Bonn, Germany. ACM Press.

[Löwe and Südholt 2006] Löwe, W. and Südholt, M., editors (2006). *Proceedings of the 5th International Symposium on Software Composition (SC 2006)*, volume 4089 of *Lecture Notes in Computer Science*, Vienna, Austria. Springer-Verlag.

[Ostermann et al. 2005] Ostermann, K., Mezini, M., and Bockisch, C. (2005). Expressive pointcuts for increased modularity. In Black, A. P., editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *LNCS*, pages 214–240. Springer-Verlag.

[Rodríguez et al. 2004] Rodríguez, L., Tanter, É., and Noyé, J. (2004). Supporting dynamic crosscutting with partial behavioral reflection: a case study. In *Proceedings of the XXIV International Conference of the Chilean Computer Science Society (SCCC 2004)*, pages 48–58, Arica, Chile. IEEE Computer Society.

[Tanter 2006a] Tanter, É. (2006a). Aspects of composition in the Reflex AOP kernel. In [Löwe and Südholt 2006], pages 98–113.

[Tanter 2006b] Tanter, É. (2006b). An extensible kernel language for AOP. In *Proceedings of AOSD Workshop on Open and Dynamic Aspect Languages*, Bonn, Germany.

[Tanter et al. 2006] Tanter, É., Gybels, K., Denker, M., and Bergel, A. (2006). Context-aware aspects. In [Löwe and Südholt 2006], pages 227–242.

[Tanter and Noyé 2005] Tanter, É. and Noyé, J. (2005). A versatile kernel for multi-language AOP. In Glück, R. and Lowry, M., editors, *Proceedings of the 4th ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2005)*, volume 3676 of *Lecture Notes in Computer Science*, pages 173–188, Tallinn, Estonia. Springer-Verlag.

[Visser 1997] Visser, E. (1997). *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam.

[Visser 2004] Visser, E. (2004). Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Springer-Verlag.