In AOSD Workshop on Open and Dynamic Aspect Languages 2006

An Extensible Kernel Language for AOP

—Reflex meets MetaBorg—

Éric Tanter
Center for Web Research, DCC, University of Chile
Avenida Blanco Encalada 2120, Santiago, Chile
etanter@dcc.uchile.cl

ABSTRACT

Reflex is a versatile kernel for AOP in Java; as such, it provides a number of core abstractions based on both structural and behavioral reflection to implement a variety of aspect-oriented approaches. In this paper, we introduce an *extensible concrete syntax for Reflex*, leading to a concise kernel language for AOP that can be extended to cover new features integrated in the kernel framework. We illustrate this extensibility with both control flow abstractions and declarative aspect composition. Furthermore, since we use the MetaBorg approach based on the syntax definition formalism SDF and the transformation language Stratego, we can embed the Reflex kernel language into the Java language and further (syntactic) extensions based on Reflex can similarly be embedded in Java+Reflex.

1. BACKGROUND

Reflex is a *versatile kernel for multi-language AOP* in Java [3]. It is a Java framework that extends Java with structural and behavioral reflective abilities. The generality of these abilities makes it a suitable backend for implementing several AOP frameworks and languages. A fundamental feature of such a kernel is its ability to detect interactions between aspects written in different languages and to propose expressive means for their resolution [2]. The model adopted by Reflex is that of explicit *links* binding a *cut* to an *action*, and characterized by a number of *attributes*. A link for behavioral reflection (we do not deal with structure here) binds a *hookset*—an intensional specification of program points to be intercepted— to a *metaobject*, *i.e.* a standard Java object corresponding to an advice instance. Further details are discussed when necessary.

Reflex is by design an extensible platform, offering a core API and a number of auxiliary libraries. One of the elements of the kernel approach to AOP we propose is that of defining (possibly domain-specific) aspect languages on top of this generic core. This paper addresses the issue of concrete syntax for the kernel itself (not the higher-level aspect languages that can be defined over it). In order to do so, we adopt the MetaBorg [1] approach, which allows unrestricted embedding of domain-specific languages into a host language. In our case, the domain-specific language is the Reflex kernel language (specific to the domain of metaobject protocols), and the host language is Java. MetaBorg consists of SDF [4] (Syntax Definition Formalism) for *modular* syntax definitions, and Stratego/XT [5] for language assimilation. We hereby focus on the SDF part only.

A feature of SDF is that it inverts productions in the syntax definition: instead of having a single production with alternatives per non-terminal (separated with "|" in BNF), SDF definitions have one production *per alternative* (see Appendix for examples). This makes it possible for an SDF module to introduce new alternatives for a given non-terminal without the need to modify the original definition, hence fostering syntactic extensibility of the language.

The next section exposes the proposed concrete syntax for Reflex, based on the familiar BNF notation (SDF is given in Appendix). We then discuss the embedding of the kernel language into Java, and illustrate the extensibility of the language with the example of control flow abstractions and declarative aspect composition.

2. CONCRETE SYNTAX FOR REFLEX

The core concrete syntax of Reflex is defined in an SDF module named `klang` (given in Appendix). It supports hookset and link declarations:

```
<ReflexDecl> ::= <HooksetDecl>
               | <LinkDecl>
```

A Reflex program is a Java program extended with a number of Reflex declarations (`ReflexDecl` in the above BNF). Reflex declarations can be located in dedicated files, or embedded within standard Java classes.

2.1 Hooksets

Hooksets are defined in Reflex by giving an *operation class* (*e.g.* message receive, send, creation, etc.) and a pair of selectors: a *class selector* to determine classes where operation occurrences should be looked for, and an *operation selector* to match precise occurrences of the given operation.

For instance, matching execution of the `calc` method in `Fib` objects is done as follows:

```
Hookset fibcall = new PrimitiveHookset(
          MsgReceive.class, new NameCS("Fib"),
          new NameOS("calc"));
```

where `NameCS` and `NameOS` are two predefined selector classes implementing the corresponding selector interface, *e.g.*:

```
class NameCS implements ClassSelector {
  String name;  /* initialized in constructor */
  boolean accept(RClass c){
    return name.equals(c.getName());
}}
```

Primitive hooksets are composable by means of operators such as union, intersection and difference.

In the kernel language, dedicated constructs for hooksets are provided. The BNF grammar for hooksets is:

```
<HooksetDecl> ::= hookset <HooksetID> { <HooksetExpr> }
<HooksetExpr> ::= <Operation> in <ClassSelExpr>
                              where <OpSelExpr>
                | <HooksetID>
                | <HooksetExpr> || <HooksetExpr>
                | <HooksetExpr> && <HooksetExpr>
                | <HooksetExpr>  -  <HooksetExpr>
```

Where `HooksetID` is a Java identifier and `Operation` is the name of an operation class (*e.g.* `MsgReceive`). `ClassSelExpr` (resp. `OpSelExpr`) is either a selector class name, or a reference to a created selector available in the scope of the declaration, or a *selection block*. Such a block is a block of Java code (delimited with square brackets) where a special variable `#cl` (resp. `#op`)[1] can be used to refer to the selector class (resp. operation) parameter.

In addition, for the often-used name-based selection, selectors can be defined using a special construct `named(n)`, where `n` is the expected name. `named(x)` for a class is equivalent to the selection block `[#cl.getName().equals("x")]`. So the `fibcall` hookset seen previously can be equivalently defined as follows:

```
hookset FibCall {
  MsgReceive in named(Fib) where named(calc)
}
```

This definition of the hookset results in the hookset being globally available (like a class), as opposed to the API-based definition, where the hookset is a simple object.

2.2 Links

A link is defined in Reflex by associating a hookset to a metaobject specification, along with a number of attributes. In particular, *call descriptors* make it possible to specify the exact invocation to the metaobject. A link declaration in the kernel language follows the grammar:

```
<LinkDecl>     ::= link <LinkID> { <LinkDeclBody> }
<LinkDeclBody> ::= <HooksetCall>+ <LinkAttrib>*
<HooksetCall>  ::= on <HooksetExpr> <CallDesc>+
<CallDesc>     ::= <Control> :
                     <JavaType>.<JavaMethod>(<Param>*)
<Control> ::= before | after | around
```

The following declares a link `LogFib` for invoking a `Logger` object each time a Fibonacci number is calculated:

```
link LogFib {
 on FibCall before : Logger.log(#this,#arg(1))
}
```

The above uses two metavariables to designate the selected piece of context information to pass as parameter to the metaobject: `#this` refers to the current `Fib` object, and the parametric metavariable `#arg` is used to extract the first argument to the `calc` method. Apart from a number of predefined metavariables for parameters (see Appendix), custom parameters can also be defined by giving a block of code that will be evaluated at runtime (*e.g.* in Sect. 4.1).

Link attributes include activation conditions, metaobject definition, initialization, and scope, as well as type constraints such as enforced types and declared type:

```
<LinkAttrib> ::= if <Activ>
               | by <MODef>
               | init <MOInit>
               | per <Scope>
               | ofType <JavaType>+
               | declared <JavaType>
```

`Activ` is a selection block defining an *activation condition*[2], which will be evaluated at runtime to determine if reification occurs or not. `MODef` states the metaobject definition scheme (*i.e.* how the reference to the metaobject is obtained):

```
<MODef> ::= new(<JavaType>)
          | shared(<JavaRef>)
          | factory(<JavaType>)
          | sharedFactory(<JavaRef>)
```

The metaobject can be obtained either by instantiating a class (`new`), or by using a reference available in the scope of the declaration (`shared`), or by invoking a newly-created factory (`factory`), or by using an existing factory object (`sharedFactory`).

Metaobject initialization and scope can be as follows:

```
<MOInit> ::= eager | lazy-safe | lazy-unsafe
<Scope>  ::= object | class | vm
```

To sum up, our `LogFib` link can be fully-defined as follows:

```
link LogFib {
 on FibCall before : Logger.log(#this,#arg(1))
 by new(Logger)
 init lazy-unsafe
 per object
}
```

This means that a new `Logger` object will be instantiated for each instance of `Fib`, and that this instantiation will occur lazily (*i.e.* the first time `calc` is called), in a non-thread safe manner (thread-safe lazy initialization requires expensive synchronization code). The same definition using the Reflex kernel API would be much less friendly:

```
Link logFib = Links.get(fibcall,
                        MODefinition.Class("Logger"));
logFib.setCall("Logger", "log",
               new Parameter[]{Parameter.THIS,
                               new Parameter.ArgN(1)});
logFib.setControl(Control.BEFORE);
logFib.setInitialization(Initialization.LAZY_UNSAFE);
logFib.setScope(Scope.OBJECT);
```

3. EMBEDDING THE KERNEL LANGUAGE

Using the MetaBorg approach and the existing infrastructure for Java syntax extension of JavaFront [1], the Reflex language is embedded in the Java language. This is important as it makes it possible to use Java variables available in the scope of an embedded Reflex declaration, and also allows syntactic extensions to Java to refer to kernel entities.

[1]We systematically use a `#` symbol for metavariables of the kernel language and embedded constructs, in order to limit conflicts with Java identifiers (`#` is illegal in Java).

[2]While activation conditions are first-class entites accessible at runtime, Reflex also supports *restrictions*, which are statically bound and cannot be changed dynamically. The SDF definition in Appendix includes support for restrictions.

3.1 Kernel Code in Java

The kernel language can be used anywhere in Java code. Thanks to this, it is possible to use Java identifiers available in the scope of the kernel code directly:

```
public class Config {
  public void initReflex() {
    Logger logger = new Logger();
    link LogFib {
      on FibCall before : Logger.log(#this,#arg(1))
      by shared(logger)
    }
} }
```

The `logger` object, created in the "Java world" can be used in the link definition to serve as the metaobject associated to the `LogFib` link; an interesting application of such an embedding is to share metaobjects between several links.

3.2 Extended Java

By defining the kernel language as an extension of Java, we also include a number of syntax extensions to Java that make it possible to reify Reflex entities as Java objects:

```
<JavaId> ::= <LinkID>#link  | <LinkID>#hookset
           | <LinkID>#activ | <LinkID>#mo
```

The following retrieves the metaobject associated to the `LogFib` link in order to change its log level:

```
  Logger logger = (Logger)LogFib#mo;
  logger.setVerbose(true);
```

Below, we retrieve a reference to the link itself, in order to deactivate it:

```
  RTLink logLink = LogFib#link;
  logLink.setActive(Active.OFF);
```

4. SYNTACTIC EXTENSIBILITY

We now illustrate the extensibility of the SDF-based definition of the kernel language with two extensions: control flow abstractions and declarative aspect composition.

4.1 Extension: Control Flow

Supporting control flow in Reflex implies first defining a link for exposing control flow information, and then making a link dependent on that control flow information (via an activation condition or a restriction). Consider the following AspectJ aspect, that traces computation of Fibonacci numbers by logging both the currently-computed value and the "top" value that was requested:

```
aspect LogFib {
  pointcut fibCall(int n): execution(* Fib.calc(int))
                           && args(n);
  pointcut topCall(int n): fibCall(n)
                    && !cflowbelow(fibCall(int))
  pointcut wormhole(int n0, int n):
       cflow(topCall(n0)) && fibCall(n);
  before(int n0, int n): wormhole(int n0, int n){
    System.out.println("Calc fib(" + n + ")" +
                       " for fib(" + n0 + ")" ;
} }
```

This aspect is interesting because it illustrates the use of several control flow pointcuts along with context exposure (the top value requested). Studying its (lower-level) expression in the kernel language shows how its basic building blocks are configured: a simple metaobject containing the advice, the `FibCall` hookset, and three links:

```
class Logger {
  void log(int n0, int n){ /* do println */ }
}
link LogFib {
  on FibCall
  before : Logger.log(
   [((CFlow)CFlowTopCall#mo).get(0)], #arg(1))
  if [((CFlow)CFlowTopCall#mo).isInside()]
}
link CFlowTopCall {
  on FibCall
  before : CFlow.enter(#arg(1))
  after : CFlow.exit()
  if [!((CFlowBelow)CFlowBelowCall#mo).isInside()]
  by new(CFlow)
}
link CFlowBelowCall {
  on FibCall
  before : CFlowBelow.enter()
  after : CFlowBelow.exit()
  by new(CFlowBelow)
}
```

`LogFib` is the main link, to which the actual advice is associated, while `CFlowTopCall` and `CFlowBelowCall` are used to expose the necessary control flow information, along with the collected values (the value of the parameter to the top call to Fibonacci). These two links make use of predefined metaobjects in the Reflex library for control flow, `CFlow` and `CFlowBelow`. They both call `enter` and `exit` methods on entry and exit of the Fibonacci calls. Internally, each control-flow metaobject maintains a thread-local depth counter (or stack if context is exposed).

The `CFlowTopCall` is subject to an activation condition based on the cflow-below information. The activation condition is specified with a selection block making use of the extended Java syntax presented in Sect. 3.2 to retrieve the metaobject and get the control flow information. `LogFib` is also subject to an activation condition, based on the control flow information of the top-level calls. In addition, the `Logger` is passed the collected value as parameter. This parameter is specified using a block (delimited by square brackets), which will be evaluated at runtime prior to invoking the metaobject.

As you can see, the code above is pretty low-level: the programmer needs to know the library classes for control flow, checking a control flow implies explicitly retrieving the exposer metaobject, the collected values are accessed by index, etc. Furthermore, there is a lot of redundancy: all control flow links have similar invocation and instantiation strategies. To enhance the situation, we introduce the following syntax extensions:

- `#in(l)` (resp. `#out(l)`) to check if the program is currently inside (resp. outside) the control flow defined by link `l`.
- a control flow call descriptor `CFlowCall`:

```
<CFlowCall> ::= expose cflow [with <ParamDecl>*]
              | expose cflowbelow [with <ParamDecl>*]
<ParamDecl> ::= <Param>:<id>
```

to declare that a control flow exposer metaobject should be used, collecting the specified parameters. Note that an identifier `id` can be associated to a collected parameter, in order to facilitate its retrieval;

- `#val(l,id)` to refer to the value of `id` collected by the control flow link `l`.

With the above syntax extension for control flow, the definition of the three links is now much more concise, and isolates the programmer from the details of the control flow library:

```
link LogFib {
  on FibCall
  before: Logger.log(#val(CFlowTopCall,n0),#arg(n0))
  if [#in(CFlowTopCall)]
}
link CFlowTopCall {
  on FibCall expose cflow with #arg(0):n0
  if [#out(CFlowBelowCall)]
}
link CFlowBelowCall {
  on FibCall expose cflowbelow
}
```

The presented syntax extensions are all contained in a single SDF module `klang-cflow` (not presented for space reason).

4.2 Extension: Declarative Composition

Reflex, as an AOP kernel, provides extensive support for managing interactions between aspects [2, 3]. To illustrate the modularity allowed by SDF when dealing with syntax extension, we show the SDF module `klang-composition` that provides concrete syntax for composition in Reflex:

```
module klang-composition
imports klang
exports
  context-free start-symbols CompilationUnit
  sorts RuleStm CompOperator
  context-free syntax
    RuleStm -> ReflexDecl
    CompOperator "(" LinkID "," LinkID ")" ";"
            -> RuleStm {cons("Rule")}
    "mutex" -> CompOperator {cons("Mutex")}
    "error" -> CompOperator {cons("Error")}
    "wrap"  -> CompOperator {cons("Wrap")}
    "seq"   -> CompOperator {cons("Seq")}
```

It introduces a new production for the `ReflexDecl` non-terminal: in addition to hookset and link declarations, the kernel language now supports rule statements (`RuleStm`). Such a statement can be used to declare a composition relation between two links: a mutual exclusion (`mutex`), a forbidden interaction (`error`), a nesting relation (`wrap`) or a sequence relation (`seq`). More composition operators can be defined [2]; providing concrete syntax for them is as simple as defining a new SDF module with the new alternative productions for the `CompOperator` non-terminal.

As an example, we can now declare ordering between the three links of Sect. 4.1 as follows:

```
seq(CFlowBelowCall, CFlowTopCall);
seq(CFlowTopCall, LogFib);
```

This ensures that the links will be applied in the order required for the evaluation of their activation conditions: `CFlowTopCall` requires information from `CFlowBelowCall`, and `LogFib` requires information from `CFlowTopCall`.

5. CONCLUSION AND PERSPECTIVES

The presented work is a first step of an on-going integration of MetaBorg and Reflex, which will eventually allow us to complete our work on the Reflex AOP kernel by allowing seamless definition and combination of new aspect languages (general and specific), addressing both syntax and semantics. The kernel language eases the use of Reflex as such, highly reducing the number of lines of code required for a given task and enhancing readability. Furthermore, it is modularly extensible. Still, the syntactic part of the Reflex kernel for newly-defined languages is not convenient: plugins for aspect languages [3] have to manage on their own the parsing and generation to Reflex code. Integrating MetaBorg and Reflex will eventually result in a plugin architecture where definition of new aspect languages, as well as extensions of existing ones, will be made much easier. This paper has concentrated on the SDF part of the story; interesting gains are expected from the integrated transformation part as well (using Stratego/XT).

Acknowledgments. The author thanks Martin Bravenboer for his help with MetaBorg, as well as Rodolfo Toledo and Leonardo Rodríguez for their feedback on the proposed concrete syntax for Reflex. Thanks to Jacques Noyé and Eelco Visser for their comments on this paper.

6. REFERENCES

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

[2] Éric Tanter. Aspects of composition in the Reflex AOP kernel. In *Proceedings of the 5th International Symposium on Software Composition (SC 2006)*, Lecture Notes in Computer Science, Vienna, Austria, March 2006. Springer-Verlag. To appear.

[3] Éric Tanter and Jacques Noyé. A versatile kernel for multi-language AOP. In Robert Glück and Mike Lowry, 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, September/October 2005. Springer-Verlag.

[4] Eelco Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.

[5] Eelco Visser. 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, June 2004.

Appendix – SDF Module for the Kernel Language

```
module klang
imports  Java-15-Prefixed
exports
  context-free start-symbols CompilationUnit
  sorts ReflexDecl HooksetDecl HooksetExpr HooksetID Operation ClassSelExpr OpSelExpr SelBlock LinkDecl LinkID
        LinkDeclBody HooksetCall CallDesc LinkAttrib IfStatic Activ Control MOInit Scope MODef Param
  context-free syntax
    HooksetDecl -> ReflexDecl
    LinkDecl    -> ReflexDecl
    ReflexDecl* -> CompilationUnit {cons("ReflexDecls")}

%%%%%%%%%%%%%%% HOOKSETS %%%%%%%%%%%%%%%%%%%%

    "hookset" HooksetID "{" HooksetExpr "}"       -> HooksetDecl {cons("HooksetDecl")}
    Operation "in" ClassSelExpr "where" OpSelExpr -> HooksetExpr {cons("Hookset")}
    HooksetID                                     -> HooksetExpr {cons("HooksetID")}
    HooksetExpr "||" HooksetExpr                  -> HooksetExpr {cons("Union")}
    HooksetExpr "&&" HooksetExpr                  -> HooksetExpr {cons("Inter")}
    HooksetExpr "-" HooksetExpr                   -> HooksetExpr {cons("Diff")}
    "(" HooksetExpr ")"                           -> HooksetExpr {bracket}
    JavaId                      -> HooksetID
    JavaTypeName                -> Operation    {cons("Operation")}
    JavaId                      -> ClassSelExpr {cons("ClsSelId")}
    JavaTypeName                -> ClassSelExpr {cons("ClsSelType")}
    SelBlock                    -> ClassSelExpr {cons("ClsSelBlock")}
    "named(" JavaTypeName ")"   -> ClassSelExpr {cons("NameCS")}
    JavaId                      -> OpSelExpr    {cons("OpSelId")}
    JavaTypeName                -> OpSelExpr    {cons("OpSelType")}
    SelBlock                    -> OpSelExpr    {cons("OpSelBlock")}
    "named(" JavaTypeName ")"   -> OpSelExpr    {cons("NameOS")}
    "[" JavaBlockStm* "]"       -> SelBlock     {cons("SelBlock")}
    "#cl"                       -> JavaExpr     {cons("ClSelVar")}
    "#op"                       -> JavaExpr     {cons("OpSelVar")}

%%%%%%%%%%%%%%% LINKS %%%%%%%%%%%%%%%%%%%%

    "link" LinkID "{" LinkDeclBody "}"    -> LinkDecl     {cons("LinkDecl")}
    JavaID                                -> LinkID       {cons("LinkID")}
    HooksetCall+ LinkAttrib*              -> LinkDeclBody {cons("LinkDeclBody")}
    "on" HooksetExpr CallDesc+ IfStatic?  -> HooksetCall  {cons("HooksetCall")}
    Control ":" JavaTypeName "." JavaMethodName "(" {Param "," }* ")"
                                          -> CallDesc     {cons("CallDesc")}
    "if-static" SelBlock            -> IfStatic   {cons("Restriction")}
    "if" Activ                      -> LinkAttrib {cons("Activation")}
    "by" MODef                      -> LinkAttrib {cons("MODefinition")}
    "init" MOInit                   -> LinkAttrib {cons("Initialization")}
    "per" Scope                     -> LinkAttrib {cons("Scope")}
    "ofType" { JavaTypeName "," }+  -> LinkAttrib {cons("Mintypes")}
    "declared" JavaTypeName         -> LinkAttrib {cons("DeclaredType")}
    JavaId         -> Activ  {cons("ActivId")}                      "before" -> Control {cons("Before")}
    JavaTypeName   -> Activ  {cons("ActivType")}                    "after"  -> Control {cons("After")}
    SelBlock       -> Activ  {cons("ActivSelBlock")}                "around" -> Control {cons("Around")}
    "eager"        -> MOInit {cons("Eager")}                        "object" -> Scope   {cons("Object")}
    "lazy-safe"    -> MOInit {cons("LazySafe")                      "class"  -> Scope   {cons("Class")}
    "lazy-unsafe"  -> MOInit {cons("LazyUnsafe")}                   "vm"     -> Scope   {cons("VM")}
    "shared(" JavaId ")"          -> MODef  {cons("SharedObj")}
    "new(" JavaTypeName ")"       -> MODef  {cons("MOClass")}
    "factory(" JavaTypeName ")"   -> MODef  {cons("Factory")}
    "sharedFactory(" JavaId ")"   -> MODef  {cons("SharedFact")}
    "#this"                       -> Param  {cons("ParamThis")}
    "#target"                     -> Param  {cons("ParamTarget")}
    "#args"                       -> Param  {cons("ParamArgs")}
    "#arg(" JavaExpr ")"          -> Param  {cons("ParamArgN")}
    "#closure"                    -> Param  {cons("ParamClosure")}
    "#method"                     -> Param  {cons("ParamMethod")}
    "#name"                       -> Param  {cons("ParamName")}
    %% more parameters... %%
    LinkID "#" "link"             -> JavaId {cons("RTLinkVar")}
    LinkID "#" "hookset"          -> JavaId {cons("HooksetVar")}
    LinkID "#" "activ"            -> JavaId {cons("ActivVar")}
    LinkID "#" "mo"               -> JavaId {cons("MOVar")}
```