Aspect-Oriented Logic Meta Programming

Kris De Volder and Theo D'Hondt
{kdvolder|tjdhondt}@vub.ac.be

Programming Technology Lab, Vrije Universiteit Brussel

Abstract. We propose to use a logic meta-system as a general framework for aspect-oriented programming. We illustrate our approach with the implementation of a simplified version of the COOL aspect language for expressing synchronization of Java programs. Using this case as an example we illustrate the principle of *aspect-oriented logic meta programming* and how it is useful for implementing weavers on the one hand and on the other hand also allows users of AOP to fine-tune, extend and adapt an aspect language to their specific needs.

1 Introduction

The notion of aspect-oriented programming (AOP) [MLTK97,KLM+97] is motivated by the observation that there are concerns of programs which defy the abstraction capabilities of traditional programming languages. At present the main idea behind software engineering is hierarchical (de)composition. Not surprisingly, current programming languages are designed from this perspective and provide mechanism of abstraction such as procedures, classes and objects, that constitute units of encapsulation specifically aimed at top-down refinement or bottom-up composition.

These abstraction mechanism however do not align well with what AOP terminology calls *cross-cutting design concerns*, such as synchronization, distribution, persistence, debugging, error handling etc. which have a wider, more systemic impact. Precisely because of their wider impact on the system, the code dealing with cross-cutting concerns can not be neatly packaged into a single unit of encapsulation. This results in their implementation being scattered throughout the source code and this severely harms the readability and maintainability of the program. Aspect-oriented programming addresses this problem by designing *aspect languages* which offer new language abstractions that allow cross-cutting aspects to be expressed separately from the base functionality. A so called *aspect weaver* generates the actual code by intertwining basic functionality code with aspect code.

Early instances of aspect languages where limited in scope and dealt only with very specific aspects in very specific contexts. The D [LK97] system for example proposes specific aspect languages to handle the aspects of synchronization and replication of method arguments in distributed Java programs. Similarly [ILGK97] describes an aspect-oriented approach in the specific context

of sparse matrix algorithms. More recently, with AspectJ [LK98], AOP has taken a turn towards a more general aspect language applicable in a broader context.

What all aspect languages have in common is that they are declarative in nature, offering a set of declarations to direct code generation. Earlier incarnations of AOP support only specific declarations suited to one particular context, e.g. synchronization in Java. Typically these stand at a high level of abstraction, avoiding specific implementation details of the weaver. Consequently they are formulated in terms of concepts linked to their context, e.g. mutual exclusion in the context of synchronization. AspectJ on the other hand provides more generally applicable but at the same time also more low-level declarations which are more intimately linked with the weaving algorithm, directly affecting code generation. Typical AspectJ declarations provide code to be executed upon entry and exit of methods, variables to be inserted into classes, etc. The universal declarative nature of aspect languages begs for a single uniform declarative formalism to be used as a general uniform aspect language. This paper proposes to use a logic programming language for this purpose. Thus, aspect declarations are not expressed by means of a specially designed aspect language but are simply logic assertions expressed in a general logic language. These logic facts function as hooks into a library of logic rules implementing the weaver.

The objective of this paper is to clearly demonstrate the advantage gained by unifying the aspect declaration language and the weaver implementation language and using a logic language for both purposes. There are two points of view from which the advantages will be discussed: the side of the AOP implementor and the side of the AOP user. The *implementor* is responsible for identifying what kind of aspects he wants to tackle. He subsequently devises a set of aspect declarations that allows expressing the aspects conveniently. Then he implements a *weaver* that takes base functionality and aspects into account and produces code that integrates them. The *user* on the other hand, is a programmer who declares base functionality and aspects by means of the special purpose languages provided by the AOP implementor.

We have called our approach *aspect-oriented logic meta programming* (AOLMP), because it depends highly on using the logic meta language to reason *about aspects*. On the one hand the logic language can be used merely as a simple general purpose declaration language, using only facts, but more importantly, it can also be used to express *queries about aspect declarations* or to declare rules which *transform aspect declarations*.

To illustrate the advantages of AOLMP from both the user's and the implementor's point of view, we will look at one aspect: the aspect of synchronization. The concern for synchronization, to ensure integrity of data accessible simultaneously by several threads, is more or less orthogonal to the program's functionality. Nevertheless, synchronization code is spread all over the program thus making it completely unintelligible. Lopes [LK97] proposed a solution for the synchronization problem using the aspect-oriented approach. She defined the aspect language COOL for expressing the synchronization aspect separately from the base functionality. We will present an implementation for COOL-like aspect

declarations, using the logic meta programming approach. Note that it is not the goal of this paper to propose a better solution to the particular problem of synchronization. Instead, the emphasis of the paper is on the technique of logic meta programming and how it relates to AOP. For reasons of clarity we therefore only provide support for a simplified subset of Lopes' declarations in our example implementation. Nevertheless, the extra flexibility of our logic meta programming approach will allow us to recover some of the omitted features and even go beyond. This results in clearer synchronization declarations because the reasoning underlying them can be made explicit through a logic program.

We used the TyRuBa system to conduct the experiments upon which this paper is based. TyRuBa was designed as an experimental system to explore the use of logic meta programming for code generation in our recent Ph.D. dissertation [DV98].

The structure of the paper is as follows. We start by briefly introducing logic meta programming and the TyRuBa system in section 2. Section 3 explains the synchronization problem and how it can be tackled using an AOP approach. This section is mostly based on Lopes' work on the COOL aspect language [LK97]. We show how it is possible to express the synchronization aspect by means of logic facts that represent COOL-like aspect declarations. Section 4 illustrates the usefulness of AOLMP from the viewpoint of an AOP user. Section 5 discusses the relevance of AOLMP for weaver implementation. Section 6 discusses related work. Section 7 summarizes the conclusions.

2 TyRuBa and Logic Meta Programming

2.1 The TyRuBa core language

The TyRuBa system is built around a core language which is basically a simplified Prolog variant with a few special features to facilitate Java-code manipulation. We assume some familiarity with Prolog and will only briefly discuss the most important differences with it. For more information about standard Prolog we refer to [DEDC96,SS94]. For more information about the TyRuBa language we refer to [DV98]. The examples throughout the paper are simple enough and sufficiently explained to be understandable to a reader not familiar with Prolog.

Quoted code blocks The most important special feature of TyRuBa is its quoting mechanism which allows pieces of Java code to be used as terms in logic programs. Quoted pieces of code may in turn contain references to logic variables or other compound terms. Pieces of quoted code containing logic variables can be used as a kind of code templates and are very useful in implementing code generators.

In the version of TyRuBa presented and used in this text the quoting mechanism is rudimentary. Basically a quoted Java term is nothing more than a kind of string starting after a "{" and ending just before the next balanced "}". Unlike a string it is not composed of characters but of Java tokens, logic variables,

constant terms and compound terms. The Java tokens are treated as special name constants whose name is equal to the printed representation of the token. The following example of a quoted Java term illustrates most kinds of quoted elements:

```
{ void foo() { Array<?El> contents  = new Array<?El>[5];
               ?El        anElement = contents.elementAt(1); }
}
```

In the above example we see a compound term “Array<?El>”. We find several name constants “contents”, “new”, “anElement”, ... There are two integer literals 5 and 1. The remainder of the tokens such as “=”, “.” and “(” are Java tokens treated as name constants with “strange names”. Note that a quoted code block may contain “{” and “}” tokens, as long as these are properly balanced. Let it be clear that a nested “{” or “}” is treated just as any other token and does *not* introduce a nested quoted code block.

The meaning of a quoted Java term in the context of a TyRuBa program is derived directly from its internal representation. A quoted Java term corresponds to a compound term of arity 1 with a special qualifier. Its single subterm is a TyRuBa list of quoted elements. The example given above is internally represented by the following compound term:

```
'{}'([void, foo, '(', ')', '{', Array<?El>, contents, '=', ... ])
```

Note that in this example the term Array<?El> is used in place of an identifier. Compound terms which occur inside blocks of quoted code are printed as mangled identifier names. This is very convenient for code generation purposes. It can be used to parameterize names for types, variables or methods. The term Array<?El> for example can be thought of as the name of a template class for representing arrays of type ?El. Also variable's and method's names can be parameterized this way. In the weaver implementation presented in section 5 we make explicit use of this feature to declare a synchronization variable per method by parameterizing the variable name with the method name.

Lexical Conventions Because terms and variables may occur inside of quoted code, TyRuBa's lexical conventions differ somewhat from Prolog's to avoid confusion. Variables are identified by a leading “?” instead of starting with a capital. This avoids confusion between Java identifiers and Prolog variables. Some examples of TyRuBa variables are: ?x, ?Abc12, etc. Consequently, any identifier, including identifiers starting with a capital, is considered to be a constant. Some examples constants are: x, 1, Abc123, etc. To avoid confusion with function or procedure calls in Java, TyRuBa compound terms are written with “<” and “>” instead of “(” and “)” as in Prolog.

2.2 Logic Meta Programming

The idea of logic meta-programming is very simple. A base program is represented indirectly by means of a set of logic propositions. The relationship between

the base program and its logic representation is concretized under the form of a code generator: a program that queries the logic data repository and outputs source code in the base language.

Logic meta-programming is thus achieved because a logic program can be thought of as representing the set of logic propositions that can be proven from its facts and rules. These facts in turn can be thought of as indirectly representing a base-language program. The full power of the logic paradigm may thus be used to describe base-language programs indirectly. This offers great potential as we will try to illustrate in the rest of this paper.

The mapping scheme between logic representation and base program may vary and determines the kind of information that is reified and accessible to meta programs. In this paper we assume a mapping that represents classes by means of facts which state that the class has certain methods, instance variables or constructors.

The presence of a variable declaration in a class is represented by a fact of the form:

```
var(?Class,?VarType,?VarName,{...declaration code...}).
```

A method declaration is asserted by a fact of the following form:

```
method(?Class,?ReturnType,?MethodName,?ArgTypeList,
   {...declaration head...},
   {...method body...}).
```

Below is an example Java class declaration and its corresponding representation as a set of TyRuBa propositions.

```
class Stack {
  int pos = 0 ;
  Stack() {
    contents = new Object[SIZE];}
  public Object peek ( )  {
    return contents[pos];  }
  public Object pop ( )  {
    return contents[--pos];  }
  ... }
```

```
class(Stack).
var(Stack,int,pos,{int pos = 0;}).
constructor(Stack,[],{public Stack()},
  {contents = new Object[SIZE]; }).
method(Stack,Object,peek,[],
  {public Object peek()},{return...}).
method(Stack,Object,pop,[],
  {public Object pop()},{return...}).
...
```

3 The Synchronization Problem and AOP

In this section we introduce the example used throughout the rest of this paper: a simple aspect language and weaver for solving the problems involved in writing synchronization code for multi-threaded Java applications. Subsection 3.1 sketches the problem and subsection 3.2 describes the aspect declarations supported by our simple weaver.

3.1 The Synchronization Problem

The problem in writing multi-threaded Java applications is that synchronization code ensuring data integrity tends to dominate the source code completely. As a result it becomes entangled and unmanageable. As an illustration of the problem, consider the implementation of a `Stack` abstract data type which is given in figure 1. The figure just lists the "bare bones" version without synchronization code. This code is simple, straightforward and easy to read.

```
class Stack  {
  static final int MAX = 10 ;
  int pos = 0 ;
  Object[] contents = new Object [ MAX ] ;

  public void print ()  {
     System.out.print("[");
     for (int i=0 ; i<pos ; i++ ) {
        System.out.print(contents[i]+" ") ; }
     System.out.print("]");  }
  public Object peek ()  {
     return contents[pos];  }
  public Object pop ()  {
     return contents[--pos];  }
  public void push (Object e)  {
     contents [pos++]=e ;  }
  public boolean empty ()  {
     return pos == 0 ;  }
  public boolean full ()  {
     return pos == MAX ;  }
}
```

Fig. 1. The "bare bones" version of the class `Stack`

The readability of the class `Stack` with synchronization code added is much worse. It is even too complicated to fit comfortably onto a single page. Therefore we will only take a look at one of the methods in it. The other methods are messed up in a similar way. Figure 2 lists the declaration of the `peek` method, complete with synchronization code.

To implement synchronization at the granularity of methods, a number of counter instance variables will be added to the `Stack` class. One such counter will be declared for each method. A counter instance variable will therefore have a name such as `BUSY_pop`, `BUSY_peek` etc. Code must be added to the start and end of each method to increment and decrement these counters. Also added to the start of the method is a "guard condition" which verifies whether the method may start executing. If the guard is not satisfied the method must wait for the guard to become true. The `peek` method for example waits until there are no

```
public Object peek ( )  {
    while ( true ) {
      synchronized ( this ) {
        if (  (  BUSY_pop == 0  ) &&  (  BUSY_push == 0  )   ) {
           ++ BUSY_peek ; break ; } }
      try { wait ( ) ; }
      catch ( InterruptedException COOLe ) { } }
    try {
     return contents [ pos ] ;  }
    finally {
      synchronized ( this ) {
        -- BUSY_peek ;
        notifyAll ( ) ; } } }
```

Fig. 2. The `peek` method with synchronization code

more threads currently executing a `push` or a `pop` method. It is obvious from figure 2 that the synchronization code completely dominates the source code: almost all of the code in the figure is synchronization code. Aspect oriented programming solves this problem by providing a special purpose language, called an *aspect language*, with which the synchronization aspect can be described separately from the base functionality. A code generator, called an *aspect weaver* takes a base program without aspects and an aspect program and generates output code integrating both.

3.2 Synchronization-aspect Declarations

Our TyRuBa weaver for generating synchronization code supports a simplified version of the COOL aspect language proposed by Lopes [LK97]. The COOL aspect language is used to specify the synchronization aspect of a Java base program. Our approach differs from the traditional AOP approach. Instead of defining a special purpose aspect language we assume that the AOP programmer provides aspect declarations under the form of TyRuBa logic facts. The weaver is consequently implemented as a library of logic rules for generating code from the facts describing the aspects and the base functionality. We defer treatment of the weaver's implementation to section 5. In this section we introduce the various aspect declarations that are recognized and supported by it.

Mutual exclusion between methods is expressed by a logic fact as illustrated by the following example.

```
mutuallyExclusive(Stack,push,pop).
```

The same kind of declaration is also used to declare that a method should never be run concurrently with itself. For example, to declare that the method `push` is not allowed to be run concurrently with itself, one asserts a fact:

```
mutuallyExclusive(Stack,push,push).
```

Declaration of mutually exclusive methods triggers the weaver to insert the appropriate guard expressions at the beginning of methods. Additional guards, other than those derived from the above synchronization declarations, may be added to a method by declaring a fact:

```
requires(?c,?m,?condition).
```

This means that the method `?m` in class `?c` may not be started unless the `?condition` expression evaluates to `true`. The following example declarations ensure that no elements are ever popped from an empty stack nor pushed onto a stack which is full.

```
requires(Stack,push,{!full()}).
requires(Stack,pop,{!empty()}).
```

Finally, declarations of facts `onEntry` and `onExit` can be used to specify synchronization related actions that have to be performed upon entry and exit of a method.

```
onEntry(?class,?method,?statements).
onExit(?class,?method,?statements).
```

4 Aspect-Oriented Meta Programming

The fundamental advantage of using logic facts to declare aspects instead of a special-purpose aspect language is that aspect declarations can be accessed and declared by logic rules. This enables what we call *aspect-oriented logic meta programming*, i.e. writing logic programs which reason about aspect declarations. This technique is useful because it allows the user to extend or adapt the aspect language. This section presents two examples of the usefulness of AOLMP from the user's point of view.

4.1 Example: Self-exclusive and Mutually-exclusive Lists

The first example is a simple extension of the aspect language which adds some syntactic sugar on top of the pairwise declaration of mutually exclusive methods. This syntactic sugar allows expressing mutual exclusion by means of lists of mutually-exclusive and self-exclusive methods, in the same style as the declarations in Lopes' system. For example, mutual exclusion of three methods can be declared using the syntactic sugar as follows:

```
mutuallyExclusiveList(Stack,[push,pop,peek]).
```

This single declaration implies that any method in the given list is mutually exclusive with any other method in the list. A similar syntactic sugar is supported to assert that methods should not run concurrently with themselves:

```
selfExclusiveList(Stack,[push,pop,print]).
```

When the list of `mutuallyExclusive` methods is long the pairwise notation becomes cumbersome and less readable because of a combinatorial explosion of pairwise combinations. We would therefore like to be able to use the list notation as syntactic sugar. It is fairly easy to add support for this. All we need to do is include two simple rules into our aspect (meta) program. The first rule expresses that two methods `?m1` and `?m2` should be declared (pairwise) mutually exclusive if they both occur together as elements in the same mutually-exclusive list `?l`.

```
mutuallyExclusive(?c,?m1,?m2) :- mutuallyExclusiveList(?c,?l),
    element(?m1,?l),
    element(?m2,?l),
    NOT(equal(?m1,?m2)).
```

Note that the above rule checks that the method `?m1` and `?m2` are not one and the same method. To actually infer that a method is mutually exclusive with itself, it must occur in a self exclusive list. This is taken care of by the following rule:

```
mutuallyExclusive(?c,?m,?m) :- selfExclusiveList(?c,?l),
    element(?m,?l).
```

The aspect program that describes the synchronization aspect of the `Stack` class, using the list-like notation is given in figure 3.

```
selfExclusiveList(Stack,[push,pop,print]).

mutuallyExclusiveList(Stack,[push,pop,peek]).
mutuallyExclusiveList(Stack,[push,pop,empty]).
mutuallyExclusiveList(Stack,[push,pop,full]).
mutuallyExclusiveList(Stack,[push,pop,print]).

requires(Stack,push,{!full()}).
requires(Stack,pop,{!empty()}).
```

Fig. 3. The `Stack` synchronization-aspect program

4.2 Example: Modifies and Inspects Declarations

This second example is a somewhat more sophisticated variation of the aspect language. The goal is to allow declaring synchronization of methods in an entirely different way. Rather than declaring which methods are mutually or self exclusive one may declare how methods modify or inspect state. This is often more convenient. For example in the `Stack`, when reasoning about what methods should be declared mutually exclusive with one another, one depends entirely on

the knowledge of which methods inspect or modify what state. It would therefore be preferable to declare this information directly and explicitly. Therefore, instead of using exclusiveness declarations, we would like to write the following:

```
modifies(Stack,push,this).
modifies(Stack,pop,this).
inspects(Stack,peek,this).
inspects(Stack,empty,this).
inspects(Stack,full,this).
modifies(Stack,print,SystemOut).
inspects(Stack,print,this).
```

This style of declarations is much more convenient than exclusiveness declarations because they relate more closely to the semantics of the methods rather than to the way that synchronization is implemented. These declaration are also clearer because they explicitly reveal which is otherwise left implicit: the reason why synchronization code must be added.

This example really highlights the advantages of declaring aspects by means of a general purpose logic language. Using logic rules which reason about and declare aspects, we can easily provide support for these alternative synchronization aspect declarations, thereby explicitizing the reasoning underlying the aspect declarations. It is also important to note that we can do this aspect-language extension without having to reimplement the weaver! We can regard the `modifies` and `inspects` aspect declarations again as syntactic sugar on top of pairwise mutual exclusive declarations. This is possible because the `modifies` and `inspects` declarations provide sufficient information to derive `mutuallyExclusive` properties. Using AOLMP we can express elegantly and concisely how both kinds of declarations relate to one another by means of two simple logic rules. The first rule expresses that any two methods are mutually exclusive if one method inspects a state modified by the other one.

```
mutuallyExclusive(?class,?inspector,?modifier) :-
   inspects(?class,?inspector,?thing),
   modifies(?class,?modifier,?thing).
```

The second rule takes care of mutual exclusion between methods which modify the same state. This rule states that modification is implicitly a kind of inspection.

```
inspects(?class,?method,?thing) :- modifies(?class,?method,?thing).
```

The examples given so far illustrate how AOLMP is useful from the AOP user's point of view. They show how expressing aspects by means of logic assertions in combination with the availability of a full-fledged logic language fosters an enormous potential for the aspect-oriented programmer. It allows him to build on top of the existing aspect declarations in order to extend or modify the aspect language. It is important to note in these examples that in order to extend the aspect language, the aspect programmer did not have to descend to the level

of the weaver's implementation. The core of the weaver's implementation has never been touched and consequently knowledge about its internal workings is not required. Extensions are simply defined as sophisticated syntactic sugar on top of already existing aspect declarations.

It might be argued that inspects/modifies declarations are less general and not able to express the synchronization aspect in some situations where exclusiveness style declarations could. This however only further highlights the power of our AOLMP approach: on the fly extensions of the aspect language can be implemented fairly easily and can be as general or as specific as a particular situation requires.

5 AOLMP and Weaver implementation

In this section we will have a look at the usefulness of AOLMP from the viewpoint of the AOP implementor. We will present the implementation of a weaver for the synchronization declarations proposed in section 3.2. For easy reference we have summarized these declarations in figure 4.

Declaration	Meaning
`mutuallyExclusive(?c,?m1,?m2)`	The method `?m1` in class `?c` should not be run concurrently with the method `?m2`.
`requires(?c,?m,?e)`	The method `?m` in class `?c` should not be started unless expression `?e` evaluates to true.
`onEntry(?c,?m,?s)`	Execute the statement `?s` upon entry of method `?m`.
`onExit(?c,?m,?s)`	Execute the statement `?s` upon exit of method `?m`.

Fig. 4. Basic synchronization-aspect declarations

5.1 Layers of Code-to-code Transformations

The architecture of the COOL weaver implementation in TyRuBa is depicted in figure 5. This is a layered architecture of code-to-code transformations. Every layer consists of logic facts describing Java source code. The Java input program with only the base functionality is parsed and turned into a set of logic facts which are inserted into the TyRuBa fact and rule base in the JCore[1] layer. A set of logic rules describes how to copy the code on the basic layer onto the COOL layer. Another set of rules describes how code is added (woven) into the COOL layer to support the aspect declarations. Finally there is one more set of rules which describes how the thus produced facts on the COOL layer should be "unparsed" into a form printable as Java source code.

[1] Named after JCore, the simplified Java language which is used in Lopes' system to express basic functionality.

Many of these rules, such as for example the copying rules and the unparsing rules just implement the general architecture and are not directly dependent on the aspect language. We will only discuss the set of rules which handles weaving of COOL aspect declarations into the COOL layer. For a more complete description we refer to [DV98].

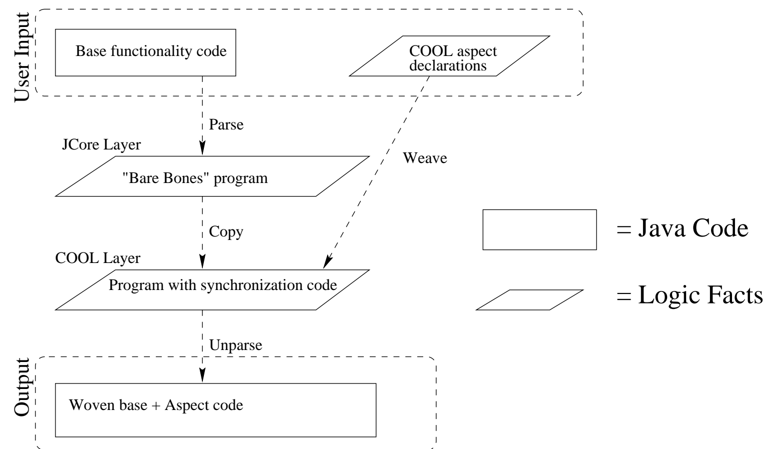

Fig. 5. The COOL code generator

Because the TyRuBa language is an experimental and simple logic language, without a module system, we could not rely on modules to divide the facts and rules into layers. We therefore partition the logic facts by adopting the convention that the first argument of every fact indicates the layer it belongs to. The JCore parser for example will insert the following fact into the rule base to indicate that the `Stack` has a `peek` method. The first argument is a symbol `JCore` indicating that this fact belongs to the `JCore` layer.

```
method(JCore,Stack,Object,peek,[],
  {public Object peek()},   //signature
  {return contents[pos]; }  //body
).
```

5.2 Synchronization-aspect Code

We will now have a look at the most important rules in the COOL weaver implementation: the rules that describe how the code on the COOL layer is generated, based on the code on the JCore layer and the synchronization aspect declarations.

Before continuing we note that part of the aspect language can be defined in terms of the more low-level features of the aspect language itself. The synchronization code for maintaining the counters could be added by means of `onEntry`

and onExit declarations. Likewise, the guard conditions that prevent methods from being started based on the value of these counters can be added by means of requires declarations. This is a very important observation because it implies that the task of implementing the weaver is greatly simplified using AOLMP. We only need to provide support for generating code for the low-level features and can then implement the higher level declarations in terms of the more low-level declarations, in a similar way as in the previous examples. We therefore start by implementing support for the more low-level aspect declarations onExit, onEntry, and requires. Afterwards we will implement the mutuallyExclusive declaration easily in terms of the more low-level declarations.

Low-level Aspect Declarations The core of the COOL code generator is very simple. Basically it merely adds some wrapper code around the body of a JCore method declaration. Below is the rule which adds wrapper code around a method in the COOL layer. This wrapper code should look familiar since it has roughly the same layout as the example synchronization code we presented for the peek method in figure 2.

```
method(COOL,?class,?Return,?name,?Args,?head,{
    while (true) {
      synchronized ( this ) {
        if (?condition) {
           ?atStart
           break; } }
      try { wait ( ) ; }
      catch ( InterruptedException COOLe ) { } }
    try {?body}
    finally {
      synchronized(this) {
        ?atEnd
        notifyAll();} }}
) :- method(JCore,?class,?Return,?name,?Args,?head,?body),
      COOL_allRequired(?class,?name,?condition),
      COOL_atStartStatements(?class,?name,?atStart),
      COOL_atEndStatements(?class,?name,?atEnd).
```

A number of auxiliary predicates computes the ?condition expression and the ?atStart and ?atEnd statement lists to be inserted into the template wrapper code.

The auxiliary predicate COOL_allRequired collects all of the conditions declared by requires aspect declarations for a certain method. All of these are combined into a conjunction, i.e. a list of Java expressions combined together by means of the Java “&&” logical “and” operator.

```
COOL_allRequired(?class,?name,?exp) :-
  FINDALL(NODUP(?cond,requires(?class,?name,?cond)),
          ?cond,?conditions),
  JavaConjunction(?conditions,?exp).
```

The meta predicate `FINDALL` is a standard Prolog feature which can be used to collect results from a given query into a list. Here it is used to collect all of the guard conditions declared in `requires` declaration into the list `?conditions`.

The meta predicate `NODUP` is a feature of TyRuBa to filter out duplicate solutions based on a comparison key. It was added to facilitate code generation. The use of `NODUP` here avoids duplicate conditions from being included more than once.

Two other auxiliary rules collect the statements to be inserted at the start and end of a method.

```
COOL_atStartStatements(?class,?name,?statements) :-
   FINDALL(onEntry(?class,?name,?stat),
           ?stat,?statements).

COOL_atEndStatements(?class,?name,?statements) :-
   FINDALL(onExit(?class,?name,?stat),
           ?stat,?statements).
```

Higher-level Aspect Declarations The rules presented in the previous section implement the core of the COOL code generator which supports the more low-level aspect declarations that insert synchronization statements and conditions at the right places into the synchronization wrapper code of a method. We can now relatively easily provide support for pairwise `mutuallyExclusive` declarations in terms of these.

First we observe that the `mutuallyExclusive` relationship is a symmetric relationship: whenever `mutuallyExclusive(?c,?m1,?m2)` holds this also implies `mutuallyExclusive(?c,?m2,?m1)`. Rather than requiring the user to declare the symmetric pairs we let the weaver implementation take care of it and declare the symmetric closure of the `mutuallyExclusive` relationship declared by the user as follows[2].

```
mutuallyExclusiveSym(?c,?m1,?m2) :-
    mutuallyExclusive(?c,?m1,?m2);mutuallyExclusive(?c,?m2,?m1).
```

The following declaration adds the guard condition that makes sure that a method `?name` is not started when another method with which it is `mutuallyExclusive` is already running[3].

```
requires(?class,?name,{BUSY<?other> == 0}) :-
   mutuallyExclusiveSym(?class,?name,?other).
```

[2] The ";" denotes a logical "or"

[3] The implementation of our simplified code generator also prohibits recursive calls from the same thread. This is usually not the intention. In Lopes' work this is patched by using a more complicated `Lock` object instead of a simple `int` counter. The `Lock` object also records which thread is locking the object and allows calls from the same thread explicitly. We could also support this more complicated locking strategy. All we need to change are the guard conditions and the declarations of the counter instance variables.

The guard expression consults a counter variable BUSY<?x> which registers how many times a method ?x has been entered. We still have to declare these variables and the onEntry and onExit code to increment and decrement the counters appropriately. The following rule adds a counter variable for every method which needs to be counted, i.e. every method that must conform to a mutuallyExclusive constraint. Note that the use of NODUP serves to avoid declaring the variable multiple times.

```
var(COOL,?class,int,BUSY<?name>,{
  private int BUSY<?name> = 0;
}) :- NODUP([?class,?name],
            mutuallyExclusiveSym(?class,?name,?other)).
```

Finally we present the rules that add administrative code for incrementing and decrementing the counter variables. Administrative code is added to every method for which a counter variable has been defined.

```
onEntry(?class,?name,{
        ++BUSY<?name>;
}) :- var(COOL,?class,int,BUSY<?name>,?declaration).
onExit(?class,?name,{
        --BUSY<?name>;
}) :- var(COOL,?class,int,BUSY<?name>,?declaration).
```

This concludes the implementation of our simplified version of the COOL aspect language and code generator. Due to lack of space we will not present the actually generated code. To get an idea of the the generated code, reexamine the peek method in figure 2. This is actually an excerpt taken from the generated code with only minor cosmetic changes to the indentation and renaming of messy "mangled TyRuBa-term identifiers" such as BUSY_Lpeek_R.

6 Related Work

6.1 Logic Meta Programming

To our knowledge, the connection between aspect-oriented programming and logic meta programming has never before been discussed or examined in the literature. The idea of logic meta programming itself, i.e. using a logic language as an expressive and powerful means to reason about programs is not new. A recent survey of the field can be found in [HG98,Bar95]. Logic programming languages are known to be good for implementing various kinds of meta programs, such as compilers, interpreters, type checkers, type inferencers etc. It's powerful unification and backtracking mechanism make it especially suitable for implementing these kinds of programs. Also, many features have been added to logic languages to facilitate meta programming. Prolog [DEDC96,CM81,SS94] for example has features to support meta programming. It offers definite clause grammars for example, a feature that facilitates the implementation of parsers. The programming language Gödel [HL94] is a declarative higher-order logic language designed

for meta programming. Lambda Prolog [FGH+90] is an extension of Prolog with unification of lambda terms. Lambda terms are an extension specifically intended to facilitate the manipulation of formulas and programs [MN87]. It is especially useful in manipulating functional programs.

A logic programming approach has also been proposed for expressing sophisticated pattern matching and verification of programs [Wuy98,Cre97,BGV90,CMR92,Min96]. The power of the logic paradigm is exploited in two major ways in these approaches: as a verification/enforcing tool (e.g. Law Governed Architectures [Min96]), or as an information gathering tool [Cre97,BGV90,CMR92], or both at the same time (e.g. SOUL [Wuy98]). Both kinds of uses of logic are complementary to AOP. All deal with code tangling. AOP tries to avoid code tangling, by allowing aspects to be expressed separately. Using logic language as an information gathering tool on the other hand, its powerful pattern matching capabilities are exploited in recovering lost information from already tangled code. As a verification tool, a logic language is used in yet another way to deal with error prone tangled code by enforcing global correctness or consistency constraints.

6.2 AspectJ

The recent developments around AspectJ [KL98] concur with our ideas in many ways. Just like our approach, AspectJ moves away from the approach of only offering a fixed set of special purpose aspect languages. Instead, it tries to capture them as particular instantiations of a more general notion of an aspect. The major difference with our approach is that AspectJ is not offering a metaprogramming language in order to achieve generality. Instead, a much more restricted extension mechanism, resembling a form of subclassing on aspects, is offered. Incidentally AspectJ also incorporates a simple form of pattern matching using wild cards as an alternative for the pattern matching power we get almost for free through unification and backtracking. Our approach is more general and more expressive than AspectJ because our extension language is a general full-fledged (logic) programming language. Of course, telling which approach is best is not as clear cut as that and other criteria besides generality and expressiveness are also important. The AspectJ team explicitly does not want to offer the full power of a real meta-programming language to the AOP user and strives to obtain a simpler and more manageable extension language.

The question remains open however whether the simplicity and manageability is worth sacrificing the expressiveness. We feel given the experimental stage of development of AOP that a more liberal more expressive formalism, such as ours, might be more suitable, at least as means of exploration and experimentation. Also, extensions and adaptation of the logic meta language towards better AOP support, such as special syntactic sugar and a scoping and module mechanism to make it more closely resemble an AspectJ like syntax might eliminate the drawbacks of a more complicated notation for simple uses altogether.

Last but not least, the `modifies/inspects` example, is at least one compellingly simple application of AOLMP for extending an aspect language. This

alone earns some merit for AOLMP as a suitable alternative for AspectJ's approach.

6.3 Reflection

Reflection is also a mechanism that can be used to deal with cross cutting. In our opinion there is however a large difference between true reflection and simpler forms of meta programming. Meta programs are programs which reason about *other* programs or aspects thereof. Reflection however, means programs which reason about *themselves*. Following the treatment of [Smi82] a reflective system has a "causally connected self representation". This means that a program has access to some kind of data structure which represents (reifies) its computational system or aspects thereof. This can be inspected or it can be acted upon. "Causally connected" means that acting upon the self representation directly affects the computational system (this is sometimes called absorption). For a more detailed explanation of this terminology and theory we refer to [Smi82,Ste94].

The self-referential nature of reflective systems makes them very complex both theoretically and with respect to implementation. Issues such as reflective overlap, meta-stability, infinite towers etc. need to be considered [Smi82,Smi84,Mae87,WF88,KdRB91,Ste94,DVS95]. The complications with reflection mainly have one common cause: its self-referential nature creates confusion between what is "meta" and what is "base". Sometimes what is "meta" can be "base" at the same time and vice versa. This confusion inevitably has its impact on the usability of reflective systems and programs because they tend to be very hard to understand. A "simple" meta system is much easier to understand and use because it has a clean separation of meta level and base level.

Our approach clearly falls into the category of "simple" meta programming. There is a very clear separation between the base program and the meta program. This can be seen easily because the base language and the meta language are actually different programming languages. The base language is Java and the meta language is a logic programming language.

This places our approach of AOLMP somewhere in between AOP and full reflection. Aspect-oriented programming is not really programming, in the sense that an aspect language is typically a restricted declarative formalism that allows asserting things about base programs, without offering the power of a programming language. Therefore aspect programs are "meta" since they are *about* programs. However, they are not real programs themselves. Our approach replaces the multitude of restricted declarative formalisms, that special purpose aspect languages typically are, by one general purpose (also declarative!) logic programming language. We however purposefully do not provide a fully reflective system because we want to keep a clear separation between meta level and base level. Mixing the two in moving towards full-fledged reflection would add confusion and complications without significantly improving the generality or expressive power of the system.

6.4 Synchronization

We have based the simple example weaver used throughout this paper on Lopes' work on D [LK97,Lop97]. The weaver presented in this text only serves as an example to illustrate the principle of AOLMP and its advantages. We did not intend to give a better solution to the particular problem of synchronization. Our example therefore is greatly simplified omitting many important features such as synchronization between multiple classes, distinction between synchronization per-class or per-instance, a more complicated and realistic locking strategy etc. However, all of these could be implemented with not too much difficulty in the logic framework we presented.

A great deal of the work involved in defining an aspect language for solving a particular problem, for example synchronization, is in deciding how exactly to describe a particular aspect. For the example of synchronization we were relieved of this task because we borrowed Lopes' design. Our approach does not offer a magical solution here: the AOP implementor still has a large responsibility in analyzing the problem and designing an aspect language for it. In this respect, the only difference is that one is designing an interface to a library of rules rather than inventing specialized syntax. Our example does illustrate how AOLMP simplifies weaver implementation once the interface to it has been designed. It also facilitates user defined variations of the aspect language. This alone may indirectly help in designing aspect languages by making it easier to experiment with alternatives.

6.5 The Transformational Approach to AOP

Work in progress by Fradet and Südholt [FS98] proposes a transformational approach to AOP. They focus on an interesting class of aspects which can be described by source to source program transformations. It is difficult to make very punctual comparisons with our approach because their work is still very much in progress. Nevertheless, we want to compare the approaches on some conceptual points because of the obvious similarities.

To a large extent our approach has a lot in common with their approach especially when considered from the AOP implementors point of view, which they seem to have focussed on. Our approach to weaver implementation in the given example is indeed a form of source to source transformation. From this point of view we simply use the logic language as a general purpose programming language which happens to be convenient as a transformation language because of its powerful pattern matching capabilities.

The transformational approach of Fradet and Südholt is based on a special purpose transformation language, specially designed for expressing transformations on abstract syntax trees. New kinds of aspect declarations consist of extensions to the abstract syntax and consequently also the concrete syntax of the language. They point out that only pattern matching on syntactic properties is not always sufficient to describe or guide the transformation and they propose to use static program analysis to remedy this.

Summarizing the above we discern three essential components in the system they envision:

1. A special purpose transformation language: To specify the aspect language. It can also serve as a generic weaver implementation language.
2. A static program analyzer construction toolkit: To deduce different kinds of static information needed in different kinds of aspects.
3. A parser generator toolkit: For creating new syntax for declaring aspects.

From the implementor's viewpoint, the added potential of our approach is found in its uniformity: the logic formalism provides a convenient substitute for all three components. A logic language can serve not only as a transformation language, but also as a general meta-programming language, well suited for implementing other kinds of meta programs, such as for example static program analyzers.

It might be useful to occasionally define special purpose syntax for some aspects. Therefore including a parser generator toolkit is not useless and could also be a useful addition to our approach. There is however no real need for it since the logic language itself can serve as a general-purpose aspect declaration language.

In many ways our work and their work is complementary in nature. Their approach is more ambitious with respect to formal underpinnings for transformational aspects, whereas ours is more directly inspired from a concrete implementation viewpoint. They also focus mostly on the aspect implementors viewpoint and consider an aspect language itself immutable once it has been defined. In contrast, this paper stresses the importance of user level programmability of aspects, for the purpose of extending the aspect language.

While it is not an central issue in this paper that the example weaver is transformational, this is still an interesting observation. It is to be expected that theoretical results from the transformational approach will be directly applicable in constructing a generic-weaver library of logic rules.

7 Conclusion

We have illustrated how an aspect language can be embedded in the logic paradigm by representing aspect declarations as logic facts. We illustrated this for a simplified version of the COOL aspect language proposed by Lopes.

That aspects are expressed by means of a full-fledged logic language represents an important advantage over using more limited special-purpose aspect languages: the logic language can serve uniformly as a formalism to declare aspects as logic facts and as a meta language for aspect-oriented logic meta programming using rules.

Aspect-oriented meta programming is useful for both users and implementors of AOP. AOP users can extend the aspect language on the fly, defining new kinds of aspect declarations. Some interesting examples where shown which do not require the user to descend to the level of the weaver implementation. This

was possible because the new declarations could be defined in terms of already existing ones. Thus, the new declarations could be defined as a kind of sophisticated syntactic sugar and implemented by means of logic rules transforming new declarations into already existing ones. This kind of programming is very interesting because it allows making the reasoning underlying the aspect declarations explicit. This was illustrated clearly by the example, where AOLMP made it possible to explicitly capture the reasoning about how modification and inspection of state is the underlying motivation for the synchronization code in a `Stack` class. The same technique is also useful at the level of the weaver implementation because often some of the aspect declarations can be defined in terms of other more low-level aspect declarations.

7.1 About the generality of the conclusions

Despite the restricted and simplified nature of the example given in this paper, we can draw some conclusions about AOLMP which are valid in a broader context because:

1. The logic language is a full-fledged (Turing complete) programming language. Consequently, it is theoretically possible to implement any conceivable weaver in it. Some weavers will be harder to implement than others, but the same is true for weaver implementation in any other language.
2. There are other known aspects which have join points very similar to the synchronization aspect such as debugging and tracing. These at least would be equally easy to implement.
3. From the user's points of view, any aspect language implemented as a library of logic rules benefits from AOLMP for building on top of the aspect language, since this is independent of the internal complexity of the library implementation itself. In an extreme case, the library may also be a logic "front end" to a weaver implemented in another programming language altogether. This would yield the advantages of user-level AOLMP without requiring a logic implementation for the weaver itself.

It is important to realize that there is no magic. Designing a good aspect language is tricky business regardless of what medium is being used to implement it. This is also true when implementing them as libraries of logic rules. For example, one of the features omitted from our simplified example weaver is inter-class synchronization. Theoretically implementing it in the logic language is not a problem. However, it is not possible to implement it by only building on top of the simplistic library exemplified in this paper. It would require the rewriting of at least a small portion of the library of rules since an inter-class synchronization policy would require a more general form of synchronization declaration which names the class together with the method:

```
mutuallyExclusive(Stack<push>,Stack<pop>).
```

The more general use of this type of declaration with methods in two distinct classes cannot be defined in terms of the building blocks provided by the simplified library. Note that it would be easy to go the other way around and provide the intra-class syntax on top of the more general inter-class syntax:

```
mutuallyExclusive(?cls<?m1>,?cls<?m2>) :- mutuallyExclusive(?cls,?m1,?m2).
```

This merely shows that aspect languages should be designed with care, just like every other programming language, library or software system.

8 Acknowledgments

I offer my gratitude to Kim Mens, Tom Tourwé, Tom Mens, Wolfgang De Meuter and Rob Nebbe for their proofreading and valuable comments on this paper and for the many inspiring discussions.

I also thank the organizers of the third AOP workshop: Cristina Lopes, Gregor Kiczales, Bedir Tekinerdogan and Wolfgang de Meuter, for their comments on an earlier version of this paper.

I thank Pascal Fradet and Mario Südholt for their comments on section 6.5 and for an interesting discussion about the subject over email.

References

[Bar95] J. Barklund. Metaprogramming in logic. *Encyclopedia of Computer Science and Technology*, 33:205–227, 1995. Also available as UPMAIL Technical Report No. 80.

[BGV90] R. Ballance, S. Graham, and M. VanDeVanter. The Pan Language-Based Editing System for Integrated Development Systems. In *Proc. 4th ACM SIGSOFT Symp. on Software Development Environments*, volume 15:6 of *ACM SIGSOFT Software Engineering Notes*, pages 77–93, 1990.

[CM81] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer-Verlag, 1981.

[CMR92] M. Consens, A. Mendelzon, and A. Ryman. Visualizing and Querying Software Structures. In *Proceedings of the 14th International Conference on Software Engineering*, pages 138–156, May 1992.

[Cre97] R.F. Crew. Astlog: A language for examining abstract syntax trees. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, Santa Barbara, California, October 1997.

[DEDC96] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard.* Springer-Verlag, New York, 1996.

[DV98] Kris De Volder. *Type-Oriented Logic Meta Programming.* PhD thesis, Vrije Universiteit Brussel, Programming Technology Laboratory, June 1998.

[DVS95] Kris De Volder and Patrick Steyaert. Construction of the Reflective Tower Based on Open Implementations. Technical Report vub-prog-tr-95-01, Programming Technology Lab, Vrije Universiteit Brussel, 1995.

[FGH+90] Amy Felty, Elsa Gunter, John Hannan, Dale Miller, Gopalan Nadathur, and Andre Scedrov. λ Prolog: An extended logic programming language. In M. Stickel, editor, *Proceedings of the Tenth International Conference on Automated Deduction (Kaiserslautern, West Germany)*, volume 449 of *lncs*, pages 754–755, Berlin, 1990. sv.

[FS98] Pascal Fradet and Mario Südholt. Aop: towards a generic framework using program transformation and analysis. In Serge Demeyer and Jan Bosch, editors, *ECOOP 98 Workshop Reader*, volume 1543 of *Lecture Notes in Computer Science*, pages 394–397. Springer Verlag, 1998.

[HG98] P. M. Hill and J. Gallagher. Meta-programming in logic programming. *Handbook of Logic in Artificial Intelligence and Logic Programming*, 5:421–498, January 1998.

[HL94] P. Hill and J. Lloyd. *The Gödel Programming Language*. MIT Press, Cambridge, MA, 1994.

[ILGK97] J. Irwin, J.-M. Loingtier, J. R. Gilbert, and G. Kiczales. Aspect-oriented programming of sparse matrix code. *Lecture Notes in Computer Science*, 1343:249–??, 1997.

[KdRB91] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[KL98] Gregor Kiczales and Cristina Videira Lopes. Tutorial 64: Aspect-oriented programming using aspectj. *OOPSLA'98 Tutorial Notes*, 1998.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97—Object-Oriented Programming, 11th European Conference*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, 9–13 June 1997. Springer.

[LK97] Cristina Videira Lopes and Gregor Kiczales. D: A language framework for distributed programming. Technical Report SPL97-007 P9710047, Xerox Palo Alto Research Center, http://www.parc.xerox/aop, 1997.

[LK98] Cristina Videira Lopes and Gregor Kiczales. Recent developments in aspectj. In Serge Demeyer and Jan Bosch, editors, *ECOOP 98 Workshop Reader*, volume 1543 of *Lecture Notes in Computer Science*, pages 398–401. Springer Verlag, 1998.

[Lop97] Cristina Videira Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, College of Computer Science, Northeastern University, November 1997.

[Mae87] Patti Maes. *Computational Reflection*. Phd thesis, Vrije Universiteit Brussel, Artificial Intelligence Lab., Brussels, Belgium, January 1987.

[Min96] Naftaly H. Minsky. Law-governed regularities in object systems, part 1: An abstract model. *Theory and Practice of Object Sytems*, 2(4):283–301, 1996.

[MLTK97] K. Mens, C. Lopes, B. Tekinerdogan, and G. Kiczales. Aspect-oriented programming. In Jan Bosch and Stuart Mitchell, editors, *ECOOP 97 Workshop Reader*, Lecture Notes in Computer Science, pages 483–496. Springer Verlag, 1997.

[MN87] Dale Miller and Gopalan Nadathur. A logic programming approach to manipulating formulas and programs. In Seif Haridi, editor, *IEEE Symposium on Logic Programming*, pages 379–388, San Francisco, September 1987.

[Smi82] Brian C. Smith. *Reflection and Semantics in a Procedural Language*. PhD thesis, MIT, January 1982. Also available as MIT/LCS/TR-272.

[Smi84] Brian C. Smith. Reflection and semantics in LISP. Report ISL-3, ACM/ Xerox PARC, Intell. Systems Lab., Palo Alto, CA, June 1984.

[SS94] Leon Sterling and Ehud Shapiro. *The Art of Prolog.* The MIT Press, Cambridge, Mass., second edition, 1994.

[Ste94] Patrick Steyaert. *Open Design of Object-Oriented Languages, A Foundation for Specialisable Reflective Language Frameworks.* PhD thesis, Vrije Universiteit Brussel, 1994.

[WF88] Mitchell Wand and Daniel P. Friedman. The mystery of the tower revealed: A non-reflective description of the reflective tower. In P. Maes and D. Nardi, editors, *Meta-Level Architectures and Reflection*, pages 111–134. Elsevier Sci. Publishers B.V. (North Holland), 1988.

[Wuy98] Roel Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proceedings of TOOLS USA'98*, 1998.