McGill University
School of Computer Science
Sable Research Group

AspectMatlab Compiler
User Manual

Sable Technical Note No. sable-2010-01

Toheed Aslam

April 24, 2010

Contents

List of Figures

List of Tables

1 Introduction

Matlab is a programming language that provides scientists with an interactive development loop, high-level array operations and a rich collection of built-in and library functions. Matlab is also a very dynamic language in which variable types are not declared, and in which new functions and scripts are loaded dynamically. Although Matlab recently incorporated object-oriented programming features, there are currently no aspect-oriented features.

We have defined an extension of Matlab, AspectMatlab, which supports the notions of *patterns* (pointcuts in AspectJ terminology), and *named actions* (advice in AspectJ terminology). An aspect in AspectMatlab looks very much like a class in the object-oriented part of Matlab. Just like classes, an aspect can have properties (fields) and methods. However, in addition, the programmer can specify patterns (pointcuts) and before, after and around actions (advice). Each action is declared with a name (unlike advice in AspectJ, which do not have names).

AspectMatlab supports traditional patterns (pointcuts) such as `call` and `execution`, but we have also concentrated on an effective design for `get` and `set` patterns which naturally deal with arrays. Loops are key control structures in scientific programs and we have developed a collection of patterns which allow one to match on loops in a variety of ways. We have also been inspired by AspectCobol [2] in that we expose join point context information via selectors that are associated with actions.

We have implemented the *amc* compiler which translates AspectMatlab source files to pure Matlab source files. The generated code can be run using any Matlab system.

2 Language Definition

Although AspectMatlab's design is mostly inspired by AspectJ, there are distinctive features of our language which are based upon two driving principles: (1) the ability to crosscut the multidimensional Matlab array accesses and loops, and (2) the ability to bind the context information from the join point shadow as part of the action declaration. While designing the syntax for the aspect constructs, we focused on achieving a couple of goals. First, enriching the patterns structure for enhanced selective matching and secondly, not to deviate from the existing language constructs for the sake of better accessibility for existing Matlab programmers.

This chapter elaborates the design of an aspect-oriented extension to a scientific programming language. We discuss the structure of an Aspect and all the constructs an aspect may contain, i.e., Properties, Patterns, Methods and Actions. It provides important details about the features of Matlab we extended and on which we based our design. We discuss supported types of patterns and actions in detail. We also describe ways to create compound user-defined patterns and how to weave actions in different orders.

2.1 Aspects

In AspectMatlab, aspects are defined using a syntax similar to Matlab classes. A Matlab class typically contains properties, methods and events. As in other object-oriented languages, properties in Matlab class encapsulate the data that belongs to instances of classes, which can be assigned default values, initialized in class constructors, and used throughout the class. Data contained in properties can be declared public, protected, or private. This data can be a fixed set of constant values, or it can be dependent on other values and calculated only when queried. Different attributes can be applied over a block of properties and property-specific access methods can be specified.

Encapsulation using methods is also a familiar concept in an object-oriented systems. MATLAB class methods are a little different as they act as an enclosing block, which can host a variety of functions. Common types of methods are ordinary functions, class constructors, class destructors and property access functions. Method blocks can be configured with different attributes, including access specifiers.

Listing 1 shows a typical MATLAB class, `myClass`, which can be used as a simple counter. This class declares a property, `count`, which has default scalar value 0. The counting functionality is provided through two functions. `incCount` increments the counter and `getCount` can be used to query the current value of count, which is returned through variable `out`. One important point to notice here is that MATLAB class methods always have the calling object automatically passed as the first argument.

Listing 1 A typical MATLAB class example

```
classdef myClass

properties
  count = 0;
end

methods

  function incCount(this)
    this.count = this.count + 1;
  end

  function out = getCount(this)
    out = this.count;
  end

end %methods

end %classdef
```

In this section, we outline the grammar of AspectMatlab in pieces as we go through related concepts and constructs. If you have a coloured version of this document, you will see that all references to productions in the McLab implementaion of the base MATLAB grammar, are given in red.

As shown in Figure 1, the base McLab `program` rule is extended to include aspects, along with functions, scripts and classes, as a program entity. Just like a MATLAB class structure, an `aspect` is named and contains a body. An aspect retains the properties and methods constructs, while adding two aspect-related constructs: patterns and actions. Patterns are formally known as pointcuts in AspectJ and are used as picking out certain join points in the program flow. AspectMatlab actions correspond to AspectJ advice, which essentially is a block of code intended to be executed at certain points in the program. This choice of terminology was intended to convey that patterns specify where to match and actions specify what to do.

Moreover, it is important to explain the `stmt_separator` non-terminal, imported from McLab. Unlike other high level languages, a MATLAB statement can be terminated in multiple ways. These

```
⟨program⟩ ::⇒⟨script⟩ | ⟨function_list⟩ | ⟨class⟩ | ⟨aspect⟩
⟨aspect⟩ ::⇒'aspect' IDENTIFIER ⟨stmt_separator⟩ ⟨help_comment⟩
        ⟨aspect_body⟩* 'end'
⟨aspect_body⟩ ::⇒
        ⟨properties_block⟩ ⟨stmt_separator⟩
        |  ⟨patterns_block⟩ ⟨stmt_separator⟩
        |  ⟨methods_block⟩ ⟨stmt_separator⟩
        |  ⟨actions_block⟩ ⟨stmt_separator⟩
```

Figure 1 Syntax of an Aspect

statement separators include the new-line, a comma or a semi-colon.

With the addition of patterns and actions, Listing 2 shows an extension to the class presented in Listing 1. `myAspect` counts how many times a function named `foo` is invoked. To achieve this functionality, we first define a pattern. Pattern `callFoo` provides us the way to specify the target join points. Once we match such join points in the source code, then we can call the corresponding action, `actCall`. This action triggers before the call to function `foo` and increments the counter.

Listing 2 A typical aspect example

```
aspect myAspect

properties
  count = 0;
end

patterns
  callFoo : call(foo);
end

methods
  function incCount(this)
    this.count = this.count + 1;
  end
end

actions
  actCall : before callFoo
    this.incCount();
  end
end

end
```

In the compiled code, an aspect is transformed into a class and the actions are translated into corresponding methods of the resulting class. As described earlier, MATLAB class methods have the invoking class object as an argument. So the methods created out of actions are also provided that object, which we named `this` for the purposes of clarity and consistency. Inside an action body, `this` should be used to interact with the properties and methods for the specific object.

We present a detailed discussion on patterns and actions in the following sections.

2.2 Patterns

Just like any other aspect-oriented language, AspectMatlab provides a variety of patterns that can be used to match basic language constructs. In addition to standard patterns such as those supported by AspectJ, a scientific programming language like Matlab possesses other important cross-cutting concerns. In Matlab, array constructs are heavily used and programs are written in the form of large functions or scripts containing many loops.

Grammar rules for patterns are presented in Figure 2. Patterns are contained inside blocks, and an aspect can contain any number of such blocks of patterns. A pattern is formed by its unique name and the pattern designators. AspectMatlab provides a number of primitive patterns targeting different constructs of Matlab. These primitive patterns can be logically combined to form the compound pattern designators. We will discuss this concept in detail in Section 2.2.6.

While providing the basic function-related patterns like **call** and **execution**, we also introduce two new sets of patterns: (1) **get/set** patterns, enabling the facility to capture array-related operations along with useful context exposure; and (2) loop patterns, which will help programmers to handle the loop iteration space and details of loop-intensive computation. AspectMatlab also supports a special within pattern, which allows us to restrict the scope of matching to certain constructs of the source code, such as functions, scripts, classes or loops.

Towards the bottom of Figure 2, we introduce some grammar rules to enable a programmar to perform selective matching. Matlab syntax allows us to make a function call, without even providing the exact number of parameters specified in the function signature. Arrays can be indexed in a similar fashion. So AspectMatlab provides a functionality to enhance the matching based on the actual parameters/indices involved. We explain the idea of selective matching in Section 2.2.3.

Moreover, matching can be performed based on the expressions containing wild card symbol "*", which results in a broader scope of matching.

A list of primitive patterns supported by AspectMatlab is presented in Table I. We discuss the different kinds of patterns in the following sections.

functions	call	captures calls to functions or scripts
	execution	captures the execution of function or script bodies
	mainexecution	captures the execution of the main function or script body
arrays	get	captures array accesses
	set	captures array assignments
loops	loop	captures execution of a whole loops
	loophead	captures the header of a loop
	loopbody	captures the body of a loop
scope	within	restricts the scope of matching

Table I List of Primitive Patterns

```
⟨patterns_block⟩ ::⇒'patterns' ⟨stmt_separator⟩ ⟨patterns_body⟩* 'end'
⟨patterns_body⟩ ::⇒IDENTIFIER ':' ⟨pattern_designators⟩ ⟨stmt_separator⟩
⟨pattern_designators⟩ ::⇒
        ⟨pattern_designators_and⟩
      | ⟨pattern_designators⟩ '|' ⟨pattern_designators_and⟩
⟨pattern_designators_and⟩ ::⇒
        ⟨pattern_designators_unary⟩
      | ⟨pattern_designators_and⟩ '&' ⟨pattern_designators_unary⟩
⟨pattern_designators_unary⟩ ::⇒
        ⟨pattern_designator⟩
      | '~' ⟨pattern_designator⟩
⟨pattern_designator⟩ ::⇒
        '(' ⟨pattern_designators⟩ ')'
      | 'set' '(' ⟨pattern_select⟩ ')'
      | 'get' '(' ⟨pattern_select⟩ ')'
      | 'call' '(' ⟨pattern_select⟩ ')'
      | 'execution' '(' ⟨pattern_select⟩ ')'
      | 'mainexecution' '(' ')'
      | 'loop' '(' ⟨pattern_select⟩ ')'
      | 'loopbody' '(' ⟨pattern_select⟩ ')'
      | 'loophead' '(' ⟨pattern_select⟩ ')'
      | 'within' '(' ⟨construct_type⟩ ','⟨pattern_select⟩ ')'
      | IDENTIFIER
⟨pattern_select⟩ ::⇒
        ⟨pattern_target⟩
      | ⟨pattern_target⟩ '(' ⟨list_dotdot⟩ ')'
⟨pattern_target⟩ ::⇒
        ⟨pattern_target_unit⟩
      | ⟨pattern_target⟩ ⟨pattern_target_unit⟩
⟨pattern_target_unit⟩ ::⇒'*' | IDENTIFIER
⟨list_dotdot⟩ ::⇒ϵ
      | '..'
      | ⟨list_star⟩
      | ⟨list_star⟩ ',' '..'
⟨list_star⟩ ::⇒'*'
      | ⟨list_star⟩ ',' '*'
⟨construct_type⟩ ::⇒'*' | 'function' | 'script' | 'loops'
      | 'class' | 'aspect'
```

Figure 2 Syntax of Patterns

2.2.1 Function Patterns

AspectJ and other aspect-oriented languages provide basic function-related cross-cutting features, which enable a programmer to track, for example, the calls made to all or some specific functions matching the specified pattern. Other places of interest in a function source code are the entry and exit points of the body.

Figure 3 shows an example of the function-related join points in the source code. The whole body of the function `main` matches an **execution** pattern, whereas every call to a function is captured by the **call** pattern.

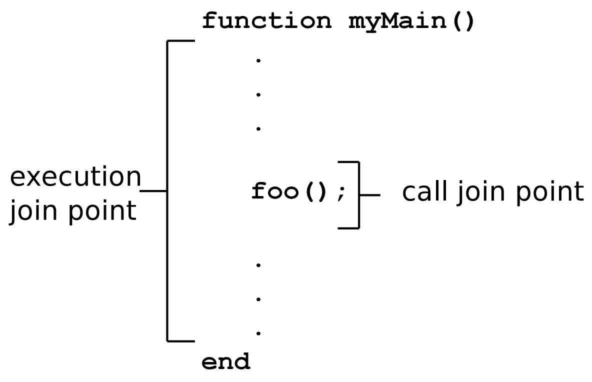


Figure 3 Function Join Points

AspectMatlab also supports both **call** and **execution** patterns, not only for functions but to cross-cut scripts as well. Because there is no specific main entry point to MATLAB programs, so we introduce a mainexecution pattern. This pattern will match the execution of the main function or script, (i.e., the first function or script executed). The function patterns given in Listing 3 show example uses, where `pCallFoo` pattern matches all calls made to the function or script named `foo` and `pExecutionMain` pattern captures the entry and exit points of the main function.

Listing 3 Function Patterns

```
patterns
  pCallFoo       : call(foo);
  pExecutionMain : mainexecution();
end
```

2.2.2 Array Patterns

AspectJ provides array pointcuts functionality. However, the pointcuts of AspectJ do not support array objects in full. When an element of an array object is set or referenced, the corresponding index values and the assigned value are not exposed to the advice. AspectJ was extended to add array pointcuts but these extensions either just work for one-dimensional arrays or they force programmers to use other pointcuts in order to be able to perform selective matching and to fetch context information.

AspectMatlab provides simple, yet powerful, patterns to capture array accesses, **get** and **set**. As shown is Figure 4, the first assignment statement is a **set** join point where `arr1` is being assigned a new value. The second statement is also a **set** join point for `arr2`, but the right hand side actually reads `arr1`. So the right hand side of the second assignment statement is a **get** join point.

Figure 5 shows an example of a more complicated **get** match. Here we have array accesses within another array access and we have to sort out the order in which all these join points are matched. We

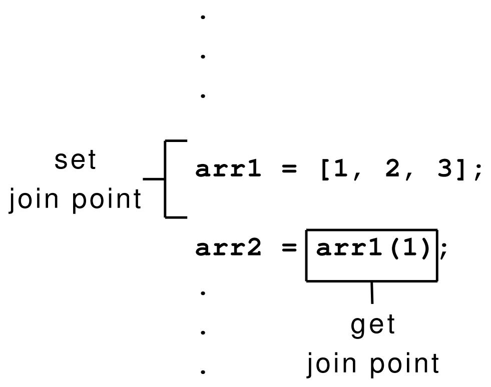


Figure 4 Array Join Points

decided to follow the evaluation order of an expression, where all the sub-expressions are evaluated before the containing expression. So, the first **get** join point in the second assigment statement is the access of `x`, followed by the second **get** join point for `y` and finally, the third **get** join point is the whole right hand side.

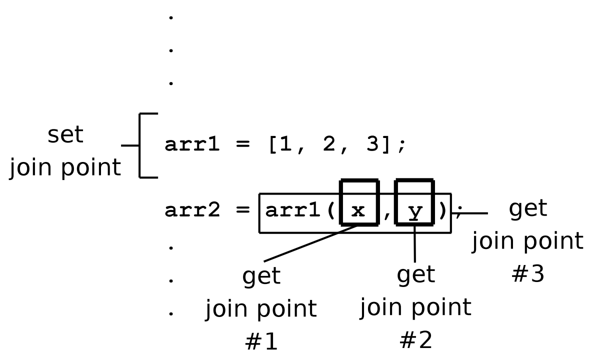


Figure 5 Array Join Points - Order

Examples of array patterns are given in Listing 4. Pattern `pGetX` matches all the join points in the source code where any array or MATLAB matrix access operation is performed. Similarly, all the write operations on the arrays can be captured using pattern `pSetX`.

Listing 4 Array Patterns

```
patterns
  pGetX : get(*);
  pSetX : set(*);
end
```

2.2.3 Selective Matching

As compared to other aspect-oriented languages, AspectMatlab eliminates the need of a separate pattern for capturing arrays and then using another pattern to specialize the matching. In MATLAB a function call does not necessarily have to provide exactly as many arguments as specified in a function signature. Also in the case of array operations, subarrays can be accessed by providing fewer dimensions than the actual dimensions of an array.

Moreover, the syntax to make a function/script call and array access in MATLAB is the same. So the pattern specification grammar was enriched to incorporate matching based upon the number of arguments involved. Section 2.2.1 and Section 2.2.2 describe simple function and array-related patterns. In this section, we provide examples of more selective matching.

As shown in the Listing 5, pattern `call2args` will match all calls, but only the ones made with two or more parameters, thus ignoring the calls with one or no parameters. If we want to match on all the arrays which are being initialized or replaced completely, pattern `fullSet` will help us achieve that.

Listing 5 Selective Matching

```
patterns
  call2args : call (*(*,..));
  fullSet   : set (*());
end
```

AspectJ also provides this facility of selective matching, but it uses separate notations for different pointcuts. The MATLAB syntax allows us to come up with a general matching notation applicable for both call/execution and get/set patterns. A list of possible use cases of such matching for the call pattern is given in Table II.

call(foo)	matches all calls to `foo` (function or script)
call(foo())	matches calls with no arguments (function or script)
call(foo(*))	matches calls with exactly one argument (function only)
call(foo(..))	matches calls with 1 or more argument(s) (function only)
call(foo(*,..))	matches calls with 2 or more arguments (function only)
	...and so on
set(arr)	matches all assignments to `arr`
set(arr())	matches assignments with no indices
set(arr(*))	matches assignments with exactly one index
set(arr(..))	matches assignments with 1 or more index/indices
set(arr(*,..))	matches assignments with 2 or more indices
	...and so on

Table II Selective Pattern Matching

2.2.4 Loop Patterns

The original AspectJ language definition did not contain any loop-related pointcuts. In MATLAB, loops are extensively used and having the ability to cross-cut the loops is equally important in such a language. AspectMatlab provides a range of poincuts for loops: **loop**, **loopbody** and **loophead**.

As shown in Figure 6, the **loop** join point presents only an outside view of the loop; because the points before and after the loop are not within the loop itself. For some applications it might be desirable to advise the loop body. Also, the loop iterators are good candidates to be advised. Because in MATLAB, loop headers are evaluated completely before the loop itself. So the **loophead** join point is not contained inside the **loop** join point.

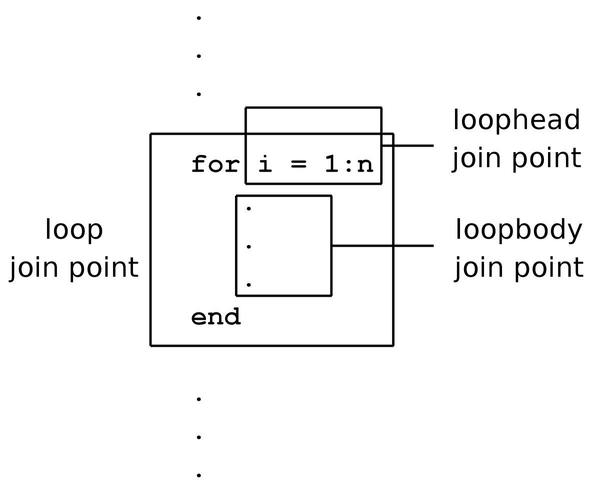


Figure 6 Loop Join Points

In aspect-oriented systems, the means of selection for a join point is, in most cases, ultimately based on the naming of some source element characterising the join point, possibly using a regular expression. For example, to advise a method call or a group of methods, the pointcut has to contain an explicit reference to some names characterising the method signatures, for instance, a pattern matching the name of the methods. Since loops can not be named in MATLAB, a name-based pattern to write a pointcut that would select a particular loop will not work.

If it is known for certain that all the loops within a function are to be advised, it would be possible in AspectMatlab to use certain scope-related pattern to restrict the loop pattern to all the loops contained in the functions picked up in the restricted scope. However, selecting only one of several loops within the same function turns out to be impossible without any further mechanism. So for the sake of loops identification, we decided to use the loop iterator variables to match a loop pattern.

Examples of simple loop patterns are given in Listing 6. All three patterns will match on all the loops, either `for` or `while`, which iterate on variable `i`.

Listing 6 Loop Patterns

```
patterns
  pLoopI      : loop(i);
  pLoopHeadI : loophead(i);
  pLoopBodyI : loopbody(i);
end
```

For example, consider the two loops shown in Listing 7, where both display the numbers from 1 to 10. Both loops match the patterns given in Listing 6.

Listing 7 Example of loop patterns

```
for i = 1:10
  disp(i);
end

i=1;
while (i≤10)
  disp(i);
  i = i+1;
end
```

2.2.5 Scope Patterns

There are certain cases in aspect-oriented systems, where some built-in language features are required to restrict the scope of matching of the patterns. For example, in AspectMatlab we use loop iterator variables to identify loops. The question might arise that names for loops iterator variables are often very general (for example, i or j), so we might end up over-matching loops unintentionally. The `within` pattern comes in very handy in such situations to restrict the scope of matching to specific constructs.

AspectMatlab supports a list of MATLAB constructs, such as `function`, `script`, `class`, `aspect` and `loops`.

Listing 8 presents examples of different cases of the within pattern. The `pWithinFoo` pattern will match every kind of join point, only inside the function `foo`. Similarly, the `pWithinBar` pattern will match every join point inside the script `bar` and the `pWithinMyClass` pattern will match every join point inside the class `myClass`. The `pWithinLoops` pattern captures all join points within all the loops. Lastly, `pWithinAllAbc` will restrict the scope to all kinds of constructs, which are named `abc`.

Listing 8 Scope Patterns

```
patterns
  pWithinFoo     : within(function, foo);
  pWithinBar     : within(script, bar);
  pWithinMyClass : within(class, myClass);
  pWithinLoops   : within(loops, *);
  pWithinAllAbc  : within(*, abc);
end
```

2.2.6 Compound Patterns

As in other aspect-oriented languages, AspectMatlab also provides a programmer the facility of creating compound patterns. Such user-defined patterns are in fact logical combination of user-defined patterns and primitive patterns given in Table I.

Examples of compound patterns given in Listing 9 display the level of flexibility a programmer can achieve in order to create different logical compounds of primitive patterns. Pattern `pCallFoo` matched all calls made to function `foo`, but only the ones from within the loops, either `for` or `while` loops. On the other hand, the pattern `pGetOrSet` will match all array read or write operations, but the ones only within the function `bar`. `pCallExec` shows a combination of an already defined pattern `pCallFoo` with a primitive pattern **execution**.

Listing 9 Compound Patterns

```
patterns
  pCallFoo  :  call(foo)  & within(loops, *);
  pGetOrSet : (get(*) | set(*))  & within(function, bar);
  pCallExec : pCallFoo | execution(foo);
end
```

Care should be taken while ANDing patterns of different kinds, because a shadow in the source code has only one specific type. For example, replacing OR with an AND in the pattern `pGetOrSet` above will result in no match, simply because an array can either be read or written to, not both at the same time.

2.3 Actions

An action is simply a named piece of code which is executed at certain points in the source code, matched by the specified patterns. An aspect can contain many actions, and as in other aspect-oriented languages, there are `before`, `around` and `after` actions.

As shown in Figure 7, an aspect can contain any number of action blocks, which in turn can host multiple actions inside them. Unlike AspectJ, actions in AspectMatlab are named. Besides the name, an action is linked to a named pattern defined in the patterns block. The type of an action specifies the weaving point of an action with respect to the join points against the pattern specified. Just like a regular MATLAB function, an action can have input parameters. These parameters are special context information which is fetched from the static shadow of each join point matched. Context exposure is described in detail in Section 2.3.1.

```
⟨actions_block⟩ ::⇒'actions' ⟨stmt_separator⟩ ⟨actions_body⟩* 'end'
⟨actions_body⟩ ::⇒
        IDENTIFIER ':' ⟨action_type⟩ IDENTIFIER ⟨stmt_separator⟩
        ⟨help_comment⟩  ⟨stmt_or_function⟩ 'end'
        |  IDENTIFIER ':' ⟨action_type⟩ IDENTIFIER ':' ⟨input_params⟩
        ⟨stmt_separator⟩ ⟨help_comment⟩ ⟨stmt_or_function⟩ 'end'
⟨action_type⟩ ::⇒'before' | 'after' | 'around'
```

Figure 7 Syntax of Actions

Simple examples of named `before` and `around` actions, which correspond to the patterns `pCallFoo` and `pExecutionMain` described in Section 2.2.1, are given in Listing 10. The action `aCountCall` will be weaved in just before each call to function `foo`. This action simply increments the `count` property defined in the properties block of the aspect. Now if we want to display the total number of calls made at the end of the program, we can use the `aExecution` action. Assuming the end of function `main` as the program exit point, `aExecution` action will be weaved in just after the whole function body.

Listing 10 Before and After Actions

```
actions
  aCountCall : before pCallFoo
    this.count = this.count + 1;
  end

  aExecution : after executionMain
    total = this.getCount();
    disp(['total calls: ', num2str(total)]);
  end
end
```

2.3.1 Context Exposure

When it comes to capturing the context of a join point, AspectCobol's [2] design doesn't rely on the use of reflection inside the advice code, as performed in AspectJ [1]. Rather, it suggests that join point reflection on the static shadow should be a part of the pointcut. The extraction of the context-specific information is described as part of the pointcut designator. We extend the idea of binding the results of desired context variables for subsequent use in the action code.

In AspectMatlab, access to the static program context that belongs to the join point is selector based. These selectors are specified along with an action definition, because an action corresponds directly to the static join point shadow. In the example below, the action `actcall`, which acts before the join points matching the pattern `call2args` given in Listing 5. It will fetch the `name` and `args` of the function call from the join point shadow.

Listing 11 Context Exposure

```
actcall : before call2args : (name, args)
  %
  disp(['calling ', name, ' with arguments(', args , ')']);
  %
end
```

Of course, a selector is only applicable depending upon the join point type. For example, the `counter` selector is only meaningful when used on a loop join point. The `args` selector fetches the array indices in case of array patterns, whereas the same selector is used to get the function arguments/parameters in case of function patterns.

A list of context selectors and their meaning with different join points is given in Table III.

2.3.2 `Around` Actions

Consider the `before` action given in Section 2.3.1, which is woven in just before the actual call to any function with 2 or more arguments. What if we want to manipulate the arguments before making such calls, or we want to add more arguments to the call, or we want to provide fewer

	set	get	call	execution	loop	loopbody	loophead
args	indices		arguments passed		loop iteration space		
obj	variable before set	variable	function handle	-	-	iterator variable	-
newVal	new array	-	-	-	-	-	loop range
counter	-	-	-	-	-	current iteration	-
name	name of the entity matched				-	-	-
pat	name of the pattern matched						
line	line number in the source code						
loc	enclosing function/script name						
file	enclosing file name						
aobj	variable name	-	-	-	-	-	-
ainput	-	input var name(s)			-	-	-
aoutput	-	-	-	output var name(s)	-	-	-
varargout	cell array variable used to return data from `around` action						

Table III Context Selectors with respect to Join Points

arguments, or we want to make such a call more than one time, or we want to call some other function instead, or we just don't want to make such function calls?

The `around` actions are the answer to all the questions. An `around` action is executed *instead of* the actual join point matched. All the valid context information can be fetched in the around action and then used accordingly. The actual join point can still be executed from within an around action, using a special `proceed` call. The `proceed` function can be called any number of times or not at all.

The `around` actions can be used with all AspectMatlab supported pattern types, except some patterns inside the script files due to MATLAB semantics. The `around` actions on such join points require these join points to be moved into a separate function, which is not possible inside a script. Unlike `before` and `after` actions, `around` actions can return data. A special MATLAB variable, `varargout`, is used for this purpose; which allows us to return multiple arguments. The `proceed` takes care of the returning arguments, but `varargout` should be set manually in case there is no `proceed`. `varargout` is a list of output values, so it needs to be made sure that it contains as many values as the original join point would return.

For example, the `around` action given in Listing 12 captures all calls to `foo` and instead calls `bar` with the same arguments. A single value returned from `bar` is set in `varargout` variable.

Listing 12 An around action without `proceed`

```
actions
  actcall : around callFoo : (args)
    % proceed not called, so varargout is set
    varargout{1} = bar(args{1}, args{2});
  end
end
```

Listing 13 shows the `around` version of the action `actcall` given in Listing 11. It simply prints out the function being called along with the arguments, before calling the `proceed`.

Listing 13 An around action with `proceed`

```
actions
  actcall : around call2args : (name, args)
    disp(['before call of ', name, 'with parameters(', args , ')']);
    proceed();
    disp(['after call of ', name, 'with parameters(', args , ')']);
  end
end
```

2.3.3 Precedence Order

As shown in Figure 8, since multiple actions can be triggered at the same join point and if more than one such actions are of the same type, we need default precedence rules for the actions:

- **before** actions are woven just before the join point. In case of multiple **before** actions, the order of the woven advice follows the exact order in which the actions were defined in source code.
- Next, **after** actions are woven just after the join point. In the case of multiple **after** actions, the order of the woven advice follows the exact order in which the actions were defined in the source code.
- Last, multiple **around** actions are woven around the join point in the exact order in which actions are defined in source code. So the outer-most of the **around** actions will be the one appearing first in the woven code and it will go around the next **around** action encountered, or the actual join point if there are no more **around** actions.

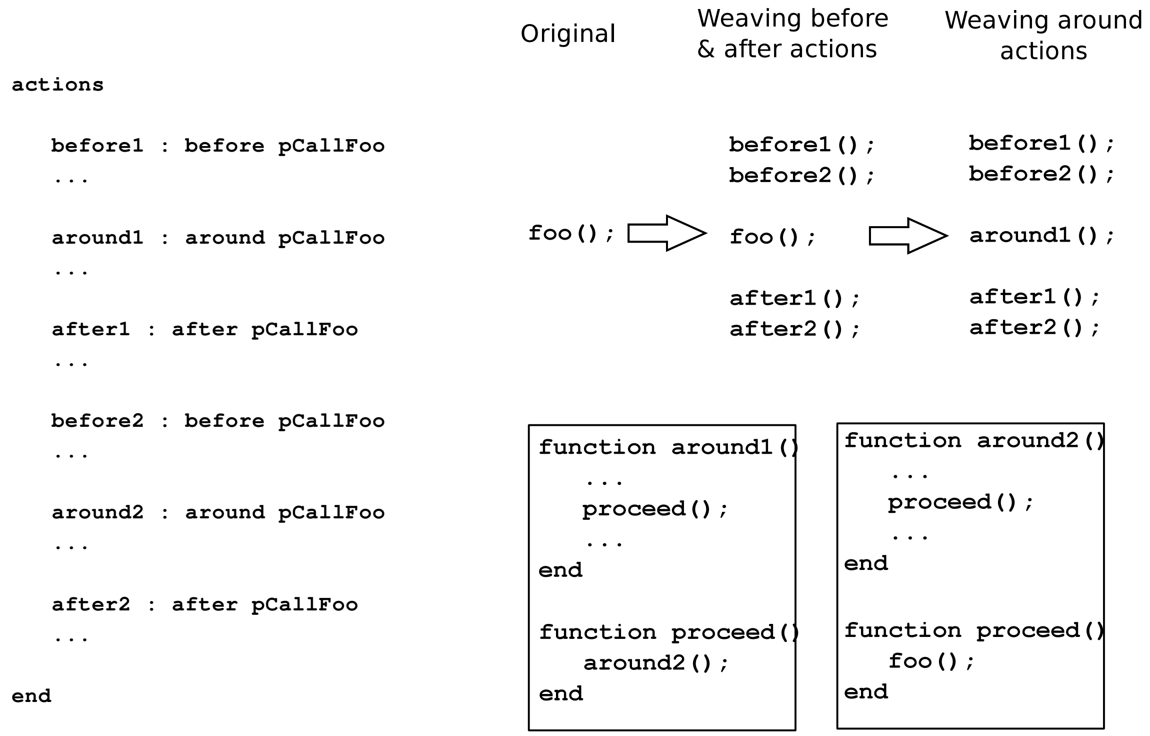
In Figure 8a, multiple actions are targeting a single **call** join point. The weaving points for a join point in the source code are shown in Figure 8b. All the **before** actions are woven just before the join point in order they are specified. All the **after** actions are woven just after the join point. The call to `foo` is replaced by the call to the first **around** action, which in turn can call the second **around** action through its proceed function, and so on.

An important point to notice here is that the default ordering rules of AspectMatlab are simpler and more restrictive than the precedence rules of AspectJ [1]. However, our action weaving strategy avoids complicated dependency rules, will not lead to any dependency cycles between actions, and is easy to comprehend from a scientific programmer's point of view. Since our actions have names, it would also be simple for us to introduce a declaration to over-ride the default ordering within each of the around, before and after groups.

2.4 Small Example

In Figure 9, we present an example of an aspect, which counts all the function calls made with at least two arguments. To do so, we need to have a **call** pattern to capture all such calls. The mainexecution pattern is used to display the number of calls made at the end of the program.

To demonstrate the application of the aspect from Figure 9, consider a small base program consisting of the simple MATLAB function given in Figure 10.



```
actions

   before1 : before pCallFoo
   ...

   around1 : around pCallFoo
   ...

   after1 : after pCallFoo
   ...

   before2 : before pCallFoo
   ...

   around2 : around pCallFoo
   ...

   after2 : after pCallFoo
   ...

end
```

(a) Actions List

```
                 before1();        before1();
                 before2();        before2();

foo();    =>     foo();      =>    around1();

                 after1();         after1();
                 after2();         after2();
```

```
function around1()
   ...
   proceed();
   ...
end

function proceed()
   around2();
end
```

```
function around2()
   ...
   proceed();
   ...
end

function proceed()
   foo();
end
```

(b) Weaving Order

Figure 8 Actions Precedence Order

The function `histo` takes one input argument `n` and returns three values `m,s,d`. Values are returned by declaring variables to be return parameters in the function header, then assigning these variables a value. This function first generates some random-sized vectors, then calls several MATLAB functions to generate a histogram, and finally computes some basic statistics.

Once compiled along with the aspect presented in Figure 9, pattern `call2args` finds only three matching join points (at lines 5, 6 and 9) where the function calls carry two arguments each. So, corresponding action function calls will be woven only at those program points. Note that the function calls with a single input argument (at lines 11, 12 and 13) do not match. Moreover, the action `actexecution` is an **after** action, so it will be woven at the end of the function.

3 Execution

A beta-release of the AspectMatlab Compiler (*amc*) is freely available to download from `www.sable.mcgill.ca/mclab/aspectmatlab`.

Once you have a copy of amc.jar, you can execute the jar directly with a list of standard MATLAB files along with AspectMatlab aspect files.

For example, one might run
```
java -jar amc.jar myFunc.m myAspect.m
```

The woven code generated by *amc* can be found in `weaved` directory in the current working directory, which can be executed by any MATLAB system.

```
aspect myAspect

properties
  count=0;
end

methods
  function out = getCount(this)
    out = this.count;
  end
  function incCount(this)
    this.count = this.count + 1;
  end
end %methods

patterns
  call2args : call (*(*,..));
  executionMain : mainexecution();
end

actions
  actcall : around call2args : (name, args)
    this.incCount();
    disp(['calling ', name, 'with parameters(', args , ')']);
    proceed();
  end
  actexecution : after executionMain
    total = this.getCount();
    disp(['total calls: ', num2str(total)]);
  end
end %actions

end %myAspect
```

Figure 9 Aspect to count all calls made with at least 2 arguments

```
function [m, s, d] = histo(n)
  % Generate vectors of random inputs
  % x1 = Normal distribution N(mean=100,sd=5)
  % x2 = Uniform distribution U(a=5,b=15)
  x1 = ( randn(n,1) * 5 ) + 100;
  x2 = 5 + rand(n,1) * ( 15 - 5 );
  y = x2.^2 ./ x1;
  % Create a histogram of the results (50 bins)
  hist(y,50);
  % Calculate summary statistics
  m = mean(y);
  s = std(y);
  d = median(y);
end
```

Figure 10 Simple MATLAB Function

3.1 Flags

amc supports the following list of flags:

- A non-aspect MATLAB file can be specified as a starting point of execution with the help of a `-main` flag. For example, `myFunc1.m` is nominated as the entry point.
  ```
  java -jar amc.jar -main myFunc1.m myFunc2.m myAspect.m
  ```
- If standard MATLAB code needs to be translated into Natlab compatible code, use `-m` flag. In the example below, all files following `-m` flag will be translated first.
  ```
  java -jar amc.jar -m myFunc.m myAspect.m
  ```
- An output directory other than the default one can be specified using a `-out` flag. For example:
  ```
  java -jar amc.jar -out output myFunc.m myAspect.m
  ```
- The version number of the *amc* can be checked using a `-version` flag. For example:
  ```
  java -jar amc.jar -version myFunc.m myAspect.m
  ```
- The usage of the *amc* can be checked using a `-h` or a a `-help` flag. For example:
  ```
  java -jar amc.jar -h
  java -jar amc.jar -help
  ```

References

[1] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP*, volume 2072, pages 327–353, 2001.

[2] R. Lämmel and K. De Schutter. What does aspect-oriented programming mean to Cobol? In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 99–110, New York, NY, USA, 2005. ACM.