AspectMatlab Reference Manual

Andrew Bodzay

May 18, 2015

1 Introduction

AspectMatlab is an extension of Matlab, which supports the notions of patterns and actions. 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). AspectMatlab supports traditional patterns commonly found in aspect languages, as well as some which target Matlab specific constructs. The purpose of this document is to introduce the various features of the AspectMatlab language, and explain how they are to be employed.

2 Aspects

In AspectMatlab , aspects were developed as an extension to object-oriented Matlab code. Object-Oriented Matlab classes are allowed to contain a `properties` block, where data that belongs to an instance of the class is defined. These properties can be defined with default values or initialized in the class constructor, and can consist of either a fixed set of constant values, or depend on other values, and be evaluated when required. Object-Oriented Matlab classes also allow for a `methods` block, which can include class constructors, property accessors, or ordinary Matlab functions. Methods and properties can be declared public, protected, or private.

AspectMatlab expands upon this by adding aspects. Aspects are an extension to the base Matlab grammar, and like a Matlab class, an aspect is a named entity, which has a body. The body of an aspect not only allows for the properties and methods constructs, but also allows for two aspect-related blocks: `patterns` and `actions`. Patterns, which are analogous to pointcuts in other aspect-oriented languages, are used to pick out sets of join points in program flow. Actions, which are analogous to advice, are blocks of code that are intended to be joined to specific points of the base program. Actions specify what should be done when code is matched by patterns.

In Figure 1 we see an example which makes use of these four features of aspects. The `properties` block defines a counter, which is initialized at its

```
aspect myAspect
properties
   counter = 0;
end

patterns
   callAdd : call(add);
end

methods
   function increment(this)
       this.counter = this.counter + 1;
   end
end

actions
   actCall : after callAdd : (name)
       this.increment();
       disp(['calling ', name]);
   end
end

end
```

Figure 1: Simple AspectMatlab example

declaration and can be used throughout the aspect. The `methods` block defines a function called increment. In the `patterns` block, we define a pattern,called callAdd, that we want to match in the base code. In this case, we match calls to the function add. Finally, the `actions` block defines an action called actCall. This action specifies that we should call the method increment after every join point in the base code which matches the pattern callAdd. It then displays the name of the function.

Patterns, which must be contained in the `patterns` block of an aspect, are formed by a unique name and a pattern designator. The pattern designator can consist of one of AspectMatlab 's several primitive patterns, each of which target specific Matlab constructs, or it can be a logical combination of them. Primitive patterns in AspectMatlab take arguments to restrict what portion of the base Matlab code should be matched. For example, in Figure 1, we see that the `call` pattern takes as a parameter 'add', meaning that it will match calls to the function add.

There are three types of actions in AspectMatlab , `before`, `around`, and `after`, which specify when, in relation to a matched join point, a piece of code should be executed. As one might expect, `before` actions are woven directly before a join point, and `after` actions are woven directly after a join point. The third type of action, `around` actions, replace the join point completely. In order to execute the join point itself when using an `around` action, a special `proceed` call exists. This call can be used in the action code to execute the original join point. Omitting this call from action code results in the original join point never being executed.

3 Patterns

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
	op	captures calls to matrix operators
Arrays	get	captures array accesses
	set	captures array assignments
Loops	loop	captures execution of whole loops
	loophead	captures the header of a loop
	loopbody	captures the body of a loop
Type	type	captures based on the class of a matrix
	dimension	captures based on the dimensions of a matrix
Annotation	annotate	captures annotations
Scope	within	captures join points within specific constructs

Table 1: Primitive AspectMatlab patterns

AspectMatlab was introduced with a variety of primitive patterns to match basic language constructs. An emphasis was made on patterns to match the cross-cutting concerns found in a scientific programming language. A list of all primitive patterns can be seen in Table 1.

3.1 Function Patterns

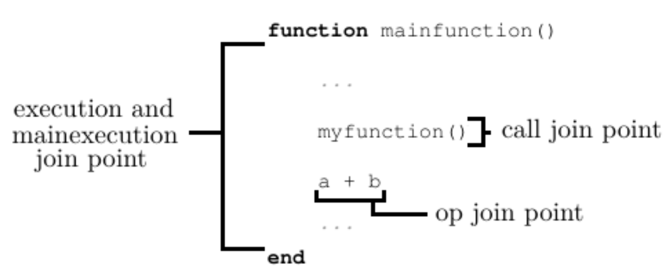


Figure 2: Function pattern join points

AspectMatlab provides multiple function-related patterns. These patterns are `call`, which matches calls to functions, `execution`, which matches function executions, `mainexecution`, which matches the execution of the main

```
patterns
    pCallFun : call(myfun);
    pMainExecution : mainexecution();
    pExecutionFun : execution(myfun);
    pAdd : op(+);
end
```

Figure 3: Function pattern examples

function, and `op`, which matches calls to MATLAB operators. The join points matched by these patterns are shown in Figure 2. The `execution` and `call` patterns take as argument the name of the function to be matched. The `op` pattern takes the operator it should match. The `mainexecution` pattern only matches the first function called, so it does not require any arguments. Examples of these patterns are shown in Figure 3. The patterns pCallFun and pExecutionFun match calls to myfun and the execution of myfun respectively. pMainExecution matches the execution of the first called function. pAdd matches any addition operations.

3.2 Array patterns

ASPECTMATLAB provides simple, yet powerful, patterns to capture array accesses and assignments. The `get` and `set` patterns both take as a single argument the variable they should match on. The `get` pattern will match whenever that variable is accessed, and the `set` pattern will match whenever that variable is assigned to. Figure 4 shows what join points will be matched by the get and set patterns. Since it is possible to have array accesses within other array accesses, patterns will be woven in the order of evaluation of the expression. Thus, in Figure 4, the access of b will be matched first, followed by d, and then finally c. Examples of the get and set patterns are shown in Figure 5. The pattern pGetX matches all accesses of the variable x. The pattern pSetAny uses the wildcard symbol, *, to indicate that it should match any assignment.

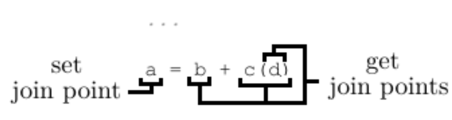


Figure 4: Array pattern join points

```
patterns
    pGetX : get(x);
    pSetAny : set(*);
end
```

Figure 5: Array pattern examples

3.3 Loop patterns

Due to their prevalence in MATLAB code, ASPECTMATLAB provides a range of pointcuts of loops: `loop`,`loopbody`, and `loophead`. As shown in Figure 3.3, the `loop` pattern matches the outside of the loop, the `loopbody` pattern matches the loops body, inside the loop, and the `loophead` pattern matches only the header of the loop where the loop iterator is evaluated. As an argument, loop patterns take the name of the iterator variable, and match only those loops with the specified iterator. Examples of loop patterns are shown in Figure 7. The pattern pGetX matches all accesses of the variable x. The pattern pSetAny uses the wildcard symbol, *, to indicate that it should match any assignment. Figure 7 lists examples of these loop patterns. pLoopX will match all loops that iterate over x, and pLoopBodyX will match the loop body of all loops that iterate over x. pLoopHeadAny, using the wildcard, *, will match the header of every loop.

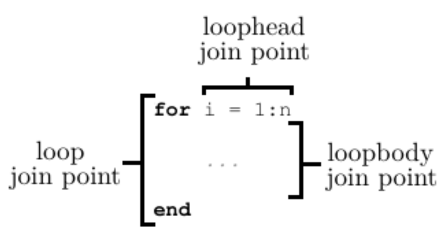


Figure 6: Loop pattern join points

```
patterns
    pLoopX : loop(x);
    pLoopBodyX : loopbody(x);
    pLoopHeadAny : loophead(*);
end
```

Figure 7: Loop pattern examples

3.4 Annotation pattern

The annotation pattern differs from other patterns in AspectMatlab in that it does not match on Matlab code itself. Instead, we allow for programmers to write annotations which take the form of structured comments in their base code. The `annotate` pattern then matches these specially formatted comments.

To specify that a particular comment should be recognized as an annotation, it is marked using the '`@`' symbol, and is followed by an identifier that gives the name of the annotation. Following the identifier is a list of arguments, whose values can be exposed as context in an action definition.

There are four types of arguments that can be exposed as context, `var` (IDENTIFIER), `str` (STRING_LITERAL), `num` (CONSTANT), and arrays of other arguments. Exposure of a `var` provides the value of that variable as context to the aspect code. `str` exposes a string, and `num` a numeric value as a double. Arrays of arguments will expose a cell array containing the context exposed by those arguments. All arguments adhere to standard Matlab syntax.

An example annotation is shown in Figure 8. It is designated as an annotation with the '`@`' symbol, has the name "type", and has 3 arguments, the variable R, the string 'double', and an array containing the values 1 and 3.

```
%@type R 'double' [1,3] %matrix of radius vectors
```

Figure 8: Example of an AspectMatlab Annotation

The `annotate` pattern matches annotations based on their name and arguments. The pattern takes as argument the name of the annotation it should match, as well as the arguments it expects the annotation to have. In Figure 9, we see pattern pAnnoType matches the annotation given in Figure 8 - it matches all annotations which have the name type, and which have a var, a char, and an array as arguments.

```
patterns
  pAnnoType : annotate(type(var, char, [*])) ;
end
```

Figure 9: Example of Annotation Pattern

3.5 Type patterns

AspectMatlab features two patterns which match based on the runtime types, the `type` pattern and the `dimension` pattern. Both of these patterns match join points corresponding to array accesses and array assignments. However, they match only when the array access or assignment meets the type criteria specified by the pattern at runtime. The `type` pattern takes as an argument the expected class type to be held, and will match array assignments and accesses which have this class. For this pattern, a class type can be one of several MATLAB defaults, such as `double`, `char`, `int32`, or it can be a user defined class type. The `dimension` pattern takes as arguments the expected dimensions held by an array at runtime, and matches when the specified dimension is met. Examples of these patterns are shown in Figure 11. pDouble matches array accesses and assignments where the array is of type double, and p2by2 matches all arrays which have dimensions 2 by 2.

```
patterns
  pDouble : type(double);
  p2by2 : dimension(2,2);
end
```

Figure 10: Example of Type Based Patterns

3.6 Within pattern

When used in conjuction with other patterns, the `within` pattern allows for restricting the scope of matching. The pattern takes two arguments, a type of construct to be matched, and the name of the construct. The pattern will match all join points within that construct. Supported constructs are `function,script,class,aspect` and `loops`. Figure **??** shows examples of this pattern. Pattern pWithinMyfun matches all join points within the function myfun, pWithinLoops matches all join points within all loops, and pWithinAllFoo matches all join points within constructs named foo (or loops which iterate over foo).

```
patterns
  pWithinMyfun : within(function,myfun);
  pWithinLoops : within(loops,*);
  pWithinAllFoo : within(*,foo);
end
```

Figure 11: Examples of Within Pattern

3.7 Compound patterns

AspectMatlab allows for the use of logical operators, and (&), or (|) and not (∼), to define more complex patterns. Examples of compound patterns are shown in Figure 12. pFooNotInFoo matches only calls to foo that are not within the function foo itself. pInt32X matches array assignments and accesses of x when x has type int32. pNestedLoop matches all loops within other loops.

```
patterns
  pFooNotInFoo : call(foo) & ¬within(function,foo);
  pInt32X : (get(x)|set(x)) & type(int32);
  pNestedLoop : loop(*) & within(loops,*);
end
```

Figure 12: Example of Compound Patterns

4 Actions

An action is a named piece of code which is executed at certain points in the source code which have been matched by the specified patterns. An aspect can contain many actions, and actions come in three varities, before, around, and after.

An aspect can have multiple actions blocks, and each action block can contain multiple actions. Actions in are named entities, and each action is linked to a pattern. The pattern can either be a reference to a named pattern specified in the patterns block, or simply the pattern itself. The type of an action determines where the woven code is inserted into the base code with respect to the matched join points. Before actions execute code before the join point, after actions execute code after the join point, and around actions eliminate the join point completely, replacing it with the woven code.

Simple examples of named before and around actions can be found in Figure 13 . The action aCountCall will be weaved in just before each call to function foo. This action simply increments a count property defined in a 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.

4.1 Context Selection

AspectMatlab allows for extraction of context-specific information about join points via the use of pre-defined context selectors. These selectors are specified along with an action definition, and allow for context-specific information to be

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

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

Figure 13: Example of before and after actions

used within action code. An example of context exposure is shown in Figure 14, where we use the name selector to expose the name of the function matched by the pattern. This information is then used to display the name of the function being called. The applicable selectors depend upon the type of join point being matched, and are shown in Figure 14.

```
actions
   actCall : after callAdd : (name)
       this.increment();
       disp(['calling ', name]);
   end
end
```

Figure 14: Example of Context Selection

	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						

Figure 15: Context Selector availability based on Join Point

4.2 Around Actions

Unlike before and after actions, an around action is executed instead of the actual join point matched. 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. Valid context information can be fetched and used accordingly, allowing one to alter the base functionality of a program. An example of an around action using a `proceed` call can be found in Figure 16, which simply prints out the function being called along with the arguments, before and after executing the function.

```
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
```

Figure 16: Example of around action

5 Using the AspectMatlab Compiler

The current release of the AspectMatlab compiler is can be downloaded at `http://www.sable.mcgill.ca/mclab/aspectmatlab/`. After obtaining a copy of compiler, it can be used to weave aspect files in one of two ways.

5.1 Execute Jar Directly

Among the included files, you can find and execute the AspectMatlab jar directly. As an example, one may run `java -jar amc.jar aspect.m matlabfunction.m`, which would apply the aspect to the function. Any number of aspects and functions may be provided, and each aspect will be woven to each function. A `weaved` directory will be placed in the current working directory, and code woven by the compiler can be found within.

When running from a terminal, AspectMatlab allows for several flags, outlined below.

-main A Matlab function file can be specified as the entry point to a program by inserting the `-main` flag before the function name.

-m By default, standard Matlab code is translated into Natlab code prior to weaving. Using the `-m` tag skips this translation.

-out The output directory can be specified using a `-out` flag, followed by the directory name.

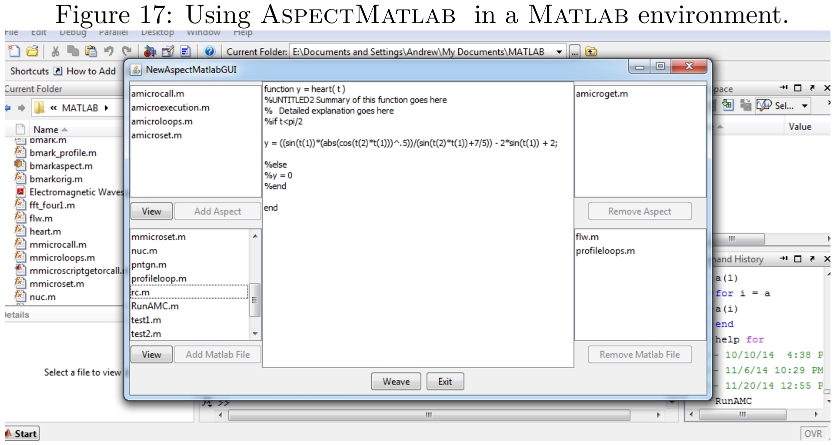
-version The version number can be checked with the `-version` flag.

-help The `-help` or `-h` flag can be can be used to describe usage of the AspectMatlab compiler.

5.2 Using AspectMatlab from within a Matlab environment

To make AspectMatlab easier to use, we have included in this release an interface that can be used from within a Matlab environment. This interface can be used to choose aspect files and Matlab functions to be woven, and allows for weaving with the push of a button. To use AspectMatlab within Matlab, simply place the `amc` directory into the working directory of your Matlab environment. Then, simply call the `runAMC` function. The interface shown in Figure 17 will be displayed.

Figure 17: Using AspectMatlab in a Matlab environment.



To add aspects, select the desired aspect from the top left box, and press the "Add Aspect" button. Added aspects will be displayed in the top right box, and can be removed with the "Remove Aspect" button. To add a Matlab file, select the desired Matlab function in the bottom left box and press the "Add Matlab File". Added Matlab functions will be displayed on the bottom right, and can be removed with the "Remove Matlab File" button. The "View" buttons can be used to preview aspects and Matlab files, displaying their contents in the center pane. Once all desired files have been selected, press the "Weave" button to run the AspectMatlab compiler with the selected aspects and Matlab files. The woven output will be placed in a `weaved` directory.

6 Remarks

This document is intended to serve as a simple user manual for those interested in learning to use AspectMatlab . To learn more, the theses by Toheed Aslam and Andrew Bodzay can be found online at `http://www.sable.mcgill.ca/mclab/projects/aspectmatlab/`. These documents detail the inner workings of the compiler and contain all relevant information concerning the features of AspectMatlab .