

AspectMatlab Reference Manual

April 13, 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.

```
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

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 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 execution mainexecution op	captures calls to functions or scripts captures the execution of function or script bodies captures the execution of the main function or script body captures calls to matrix operators
Arrays	get set	captures array accesses captures array assignments
Loops	loop loophead loopbody	captures execution of whole loops captures the header of a loop captures the body of a loop
Type	type dimension	captures based on the class of a matrix 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

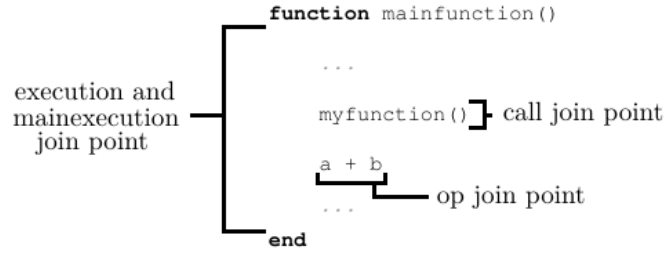


Figure 2: Function pattern join points

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 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

```

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

```

Figure 3: Function pattern examples

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.

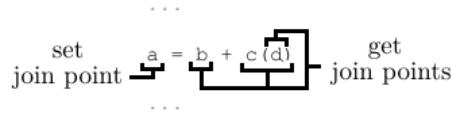


Figure 4: Array pattern join points

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

```

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

```

Figure 5: Array pattern examples

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.

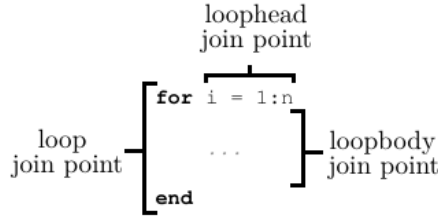


Figure 6: Loop pattern join points

```

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 conjunction 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

dasda

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.

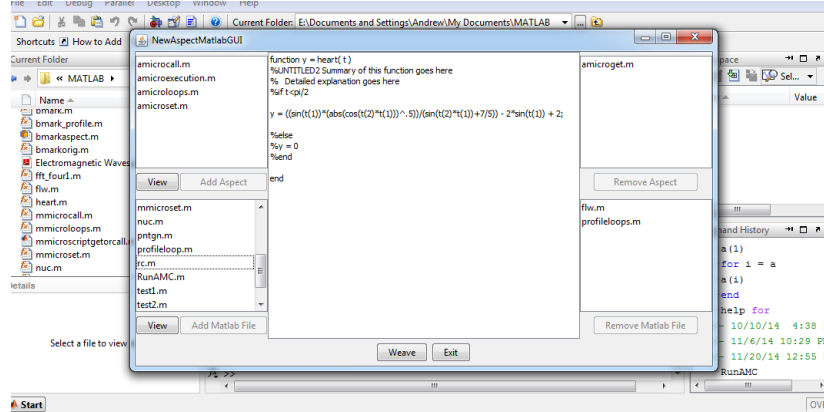
- main** A MATLAB function file can be specified as the entry point to a program by inserting the **-main** flag before the function name.
- m** By default, standard MATLAB code is translated into Natlab code prior to weaving. Using the **-m** tag skips this translation.
- out** The output directory 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 13 will be displayed.

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

Figure 13: Using ASPECTMATLAB in a MATLAB environment.



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 .