# A Domain-Specific Aspect Language for Transforming MATLAB Programs

João M. P. Cardoso
Dep. Engenharia Informática
Faculdade de Engenharia (FEUP)
Universidade do Porto
Porto, Portugal
jmpc@acm.org

Pedro C. Diniz
Dep. Engenharia Informática
UTL/IST/INESC-ID
Lisboa, Portugal
pedro.diniz@ist.utl.pt

Miguel P. Monteiro
CITI, Dep. Informática
Universidade Nova de Lisboa
Monte de Caparica, Portugal
mmonteiro@di.fct.unl.pt

João M. Fernandes, João Saraiva
Dep. Informática / CCTC
Universidade do Minho
Braga, Portugal
{jmf,jas}@di.uminho.pt

## ABSTRACT

Aspect-oriented programming enables software developers to augment programs with information out of the scope of the base language while not hampering the code readability and thus its portability. MATLAB is a popular modeling/programming language that can significantly benefit from aspect-oriented programming features. Crosscutting concerns include various forms of optimization-motivated configurations, namely the restriction on allowed data types/values, monitoring of specific execution facets, such as dataset sizes and the assignment of specific values to variables. This paper describes the main concepts of a domain-specific aspect language (DSAL) for specifying transformations in MATLAB to facilitate the experimentation of alternative variants of a core code. In the proposed language Aspect modules are structured in three sections: *intersections* (select) equivalent to AspectJ poincuts, *actions* (apply) equivalent to advice, and *conditions* (when) that control triggering of actions. Two key features proposed are aspect composition strategies to support for specifying particular sequences of aspects and parameterization of aspects for supporting the definition of parameters that specialize aspects.

## Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features. D.2.3 **[Software Engineering]**: Coding Tools and Techniques. D.1 [**Programming Techniques**].

## General Terms

Performance, Experimentation, Languages.

## Keywords

Aspect-Oriented Programming, Strategic Programming, Domain-Specific Languages, MATLAB.

## 1. INTRODUCTION

MATLAB [9] is an interpreted, imperative programming language intended to operate primarily on matrix-shaped double precision data types. Available MATLAB features and packages help programmers to focus on problem solving and allow high expressiveness when dealing with matrix computations, thus contributing to enhanced productivity. It is widely used in scientific computing, control systems, signal processing, image processing, system engineering and simulation. MATLAB relies heavily on array variables and double precision data types. However, the flexibility of its interpretative nature also hinders performance, forcing programmers to develop reference versions of the program functionality in languages such as C/C++, especially when targeting embedded systems. When doing so, programmers effectively freeze important decisions relating to specific data types and program structure thereby forsaking most of MATLAB's flexibility. The complexity resulting from sophisticated specializations is exacerbated by changing program requirements (e.g., power vs. performance) and target architecture features (e.g., CPU vs. GPU).

Available MATLAB features and packages help programmers to focus on problem solving and allow high expressiveness when dealing with matrix computations, thus contributing to enhanced productivity. However, when it comes to evaluate specific features, such as exploring non-uniform fixed-point representations, monitoring specific variables during a timing window, or including handlers to watch specific behaviors, the programmer is overwhelmed by cumbersome, error-prone and tedious tasks. Each time new features are needed, invasive changes on the original code are required, as well as the insertion of new code. These problems are felt in issues related to implementation criteria, such as those arising in embedded system applications (e.g., low instruction counts or memory footprint), as MATLAB can be approached as a specification, rather than as an implementation language.

In previous work [4], we proposed aspect-oriented features to MATLAB to support monitoring of variable values, testing the use of alternative implementations, handling of specific conditions and specifying data types. Current efforts focus on augmenting the MATLAB programming methodology by using a DSAL with richer aspect-oriented concepts as described in this paper. The proposed concepts allow the exploration of specific features within the system's design and implementation space, debugging and monitoring, and specification of programmers' knowledge about an algorithm not directly captured in the MATLAB program structure. This approach enables a single manual version of the MATLAB specification to be used throughout the entire development cycle without the need to maintain multiple versions,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*DSAL'10*, March 16, 2010, Rennes, France.
Copyright 2010 ACM 1-58113-000-0/00/0004…$5.00.

as is currently the case. We believe this separation helps the development, simulation, exploration and implementation phases.

This paper describes the use of aspects to convey information regarding characteristics of the programming elements, such as the actual values of variables, as well as to specify transformation to be applied to specific programming constructs. We propose aspect modules expressed in a DSAL based on the key concepts of joinpoint selection (select) composition/action (apply) and conditional binding (when), through which programmers provide knowledge of additional facets to a compiler/run-time system that otherwise would be hard or impossible to derive from the original program.

A simple use case of the supported aspects relates to shapes of array variables and base types, which can be specified through aspects for specific points of the program and/or depending on specific variable values or execution points. The approach also allows the specification of source-level program transformations such as loop unrolling or function inlining applied to specific input values or sizes of variables. We also propose an abstract strategy mechanism to enable programmers to explore the optimization space by applying a sequence of program transformations subject to values resulting from a specific aspect execution. Simple examples include strategies that transform a given section of the code only when the shape of an input variable has a specific value or the size of one of its dimension exceeds a given value.

The original base program remains free of language enhancements and base sources remain syntactically correct. The proposed DSAL enables programmers to retain the advantages of a single source program representation, while exploring a wide range of specific solutions at reduced programming and maintenance costs.

The rest of the paper is organized as follows. Section 2 describes the main concepts and language features proposed. Section 3 briefly discusses implementation issues. Section 4 compares this approach to related work. Finally, we conclude in Section 5.

## 2. ASPECT CONCEPTS AND DESIGN

*Aspect modules* are the main components of the proposed DSAL. Each aspect module is structured as illustrated in Figure 1(a) and can have several *select-apply-when* code sections – all are considered when executing that aspect. Aspects may have input arguments and return output information. Supported inputs and outputs include parameters to specialize an aspect, clauses to constrain the scope of an aspect intersection to a set of intersections previously specified by another aspect, and variables. The aspect programmer can specify the order with which aspects are to be executed. Different sequences can be structured as strategies.

Figure 1(b) presents an example of an aspect module, which is responsible to insert specific MATLAB code (to report a warning message based on the evaluation of a condition) before each use of the variables *sum* and *A*. Figure 2(b) shows the resulting code of applying this aspect to the MATLAB code in Figure 2(a). A more generic aspect module is illustrated in Figure 3 producing the same result as the aspect in Figure 1(b), if applied as *warning_too_big({sum,A}, 10000);* to the code in Figure 2(a).

Next, the main concepts are described, namely joinpoint selections, advice-like actions, conditions, and strategies.

```
aspect <name>
  (input: ...)?
  (output: ...)?
  (select: ... end
  apply: ... end
  (when: ... end)?)+
end <name>
```

(a) the structure of an aspect module.

```
aspect warning_too_big
  select:
    all reads <var a1> in {sum, A}
  apply:
    insert { if <a1.name> >= 10000 warning
    ('<a1.name> too big! %f', <a1.name>);
    end }:: execute before
end warning_too_big
```

(b) example of an aspect module.

**Figure 1. Aspect module, the main component of the language.**

```
...
for j = 1:1:N
  sum = sum +
        A(j) *
        B(j+N);
end
outa(i) = sum;
...
```

(a) piece of MATLAB code.

```
... for j = 1:1:N
  if sum>=10000 warning ('sum too big! %f',sum);
end
  if A(j)>=10000 warning ('A(j) too big! %f',A(j));
end
  sum = sum + A(j) * B(j+N);
end
if sum>=10000 warning ('sum too big! %f',sum);
end
outa(i) = sum; ...
```

(b) MATLAB code with logging code (underlined and in italic).

**Figure 2. Code inserted for logging if certain variables exceed a specific value (10000).**

```
aspect warning_too_big
  input: <var *>, <const c1>
  select:
    all reads <var a1> in {<var *>}
  apply:
    insert { if <a1.name> >= <c1.value> warning ('<a1.name> too
    big! %f', <a1.name>); end }:: execute before
end warning_too_big
```

**Figure 3. A parameterized aspect component.**

### 2.1 The Joinpoint Model

Unlike in other aspect-oriented approaches, including AspectJ [5], joinpoints are not restricted to method invocations, object instantiations, and variable accesses. Joinpoints can be identified by a *name* bound to an identifier (of a variable or function), a broader characteristic (e.g., *all variables*, *all reads of certain variables*, *all invocations of a function*), or by an intersection pattern. Figure 1(b) illustrates an aspect component that intersects MATLAB code in all the read operations of variables *sum* and *A*. Figure 3 illustrates an aspect with the same functionality but able to receive a set of variables for intersection. In addition, we use annotation-like tags embedded in MATLAB comments to specify joinpoints, through the convention that such tags start with '%@', e.g., %@here1, %@loop1. The symbol "%" is the beginning of a comment line in MATLAB and consequently the resulting annotated MATLAB code remains syntactically correct.

Intersections include a scheme to define *intersection patterns* by allowing lexical matching and exact/approximate syntactic matching. Figure 4 shows an example of a pattern matching specification of a corresponding intersection that performs loop unrolling with an unrolling factor of 2. Here <var *a1*> matches the loop control variable and <body> the set of statements for the original loop body. In the transformed code occurrences of the parameter variables *a1* are replaced by an expression "<*a1*> + 1".

## 2.2 Actions as Advice

Actions equivalent to AspectJ advice are associated with one or more joinpoints and can be of three usual kinds with respect to the action: insert, replace, and remove. At a particular joinpoint, the associated action is activated and executed if enabled by its *when* clause The three usual types are supported: "around" (over a joinpoint, i.e., the action replaces the code associated to that joinpoint), "before" (action is executed before the code in that joinpoint), and "after" (action is executed after the code in that joinpoint). Recall that this joinpoint can be either a high-level construct or a single occurrence of a variable identifier.

```
...
for i=1:1:100
  A(i) = B(i) + 1;
end
...
```

(a) MATLAB base code.

```
aspect loopTransf
  select: {
    for <var a1> = 1:1: <const integer c1>
        <body>
    end
  } :: position innermost
  apply: insert {
    for <a1.name> = 1:2:<c1.value>-1
        <body>
        <body(replace <a1.name>
            with <a1.name>+1)>
    end } :: execute around
  when: static {
    if <c1.value> % 2  == 0 }
end
```

(b) aspect module with intersection pattern.

```
...
for i=1:2:99
    A(i) = B(i) + 1;
    A(i+1) = B(i+1) + 1;
end
...
```

(c) resulting code after weaving base and aspect

**Figure 4. Example of an intersection mechanism, using pattern matching, and an action controlled by a static condition.**

## 2.3 Triggering Conditions

*Conditions* are enablers/disablers of the execution of actions. Actions without conditions are always executed. Figure 4 and Figure 5 present examples of static and dynamic conditions, respectively. In each case, the condition evaluates if the upper bound of the iteration range is a multiple of 2. In the static condition, the action (i.e., the code transformation) is executed only if this condition evaluates to true. The dynamic condition instructs the weaver to include the original intersected code in the output code and the modified code according to the action, selecting between them based on the evaluation of the condition.

## 2.4 Aspect Strategies

We allow programmers to specify a specific sequence for the application of the actions associated with through a *strategy*. For example, the aspect strategy "*A: aspect1 → aspect2 → aspect3*", specified in Figure 6(a), means that the weaver must first execute aspect1, then aspect2, and finally aspect3. Each aspect from the sequence may modify code and new modifications may follow previous modifications. Although finding the appropriate and correct strategy is an interesting research topic, in this work we focus on the programming support for aspect strategies.

We use an imperative style for specifying aspect strategies. Mechanisms are provided to perform typical control flow. This strategic programming must deal with the following issues:

- recursive application of an aspect while a given condition holds (e.g., an aspect to unroll loops, based on a pattern, can be invoked recursively in the nested loop structure until no further modification occurs),
- execution of different sequences in paths enabled by conditions,
- use of loops to repeat sequences of aspects, and
- passing data between aspects.

```
when:
  dynamic {
    if <a2.name> %
    2  == 0
}
```

(a) dynamic condition.

```
... if N % 2 == 0
  for i=1:2:N
    A(i) = B(i) + 1;
    A(i+1) = B(i+1) + 1;
  end
else % if pattern is not matched
  for i=1:1:N
    A(i) = B(i) + 1;
  end
end ...
```

(b) example of code after weaving.

**Figure 5. A dynamic condition and the result (considering the MATLAB code of Figure 4(a)).**

Aspect strategies define possible flows of aspects, defined in *aspect management units* (examples in Figure 6). For each call of an aspect, information can be returned to the aspect management unit, which may consist of a set of aspect attributes for each intersection of the aspect in a given call.

```
apply: A
strategy A
  aspect1;
  aspect2;
  aspect3;
end A
```

(a) strategy for a sequence of aspects.

```
apply: B
strategy B
   do
      a1=aspect1;
   while(a1.modified);
end B
```

(b) an aspect repeated while a condition holds

**Figure 6. Examples of aspect strategies.**

The scope for intersection of an aspect can be a set of regions of code given by the intersection of a previous aspect. This is specified by inputting to an aspect the intersection region as occurred in a previous aspect, as illustrated in the following example: *a1=aspect1 → aspect2(a1.intersection)*

The two examples in Figure 6 illustrate strategies used by the aspect management unit. Example (a) illustrates an aspect strategy for defining a sequence of three aspects. Example (b) illustrates an aspect strategy where an aspect is repeated while a certain condition holds.

## 2.5 Reference Variables

The intersection subsection (*select*) of aspect modules can define variables to be used in the other two sections (*apply* and *when*). Thus, base code can be modified/specialized by assigning different values to variables present in the code. If a code segment <body> uses a variable defined as <var *a1*>, the reference *a1* can be used to modify or replace the name of the variable referred by *a1* as illustrated in Figure 4, where the variable name is concatenated with "+1". These variables have attributes that can be used in the *apply* and *when* sections. Attributes are identified by the name of the variable followed by '.' and the attribute name (e.g., "*a.name*" for the variable *a*).

One important feature of the above variables is that they can modify other inner variables. An example is the code *insert{p1(replace <c1.value> with "100")}* in which "*p1*" identifies a code pattern such as the one in the example in Figure 4. In this case, code related to pattern "*p1*" is inserted in joinpoints specified by the select section of the aspect, and constant "*c1*" in the pattern is replaced by "100".

Reference variables are also a mechanism to manage differences in the actions performed by the same aspect module. For instance, they allow different values for the same pattern on the basis of the point in the program where the pattern intersects.

## 2.6 Generalization of Aspects

Aspect generalization, in the sense of parameterization, is supported as in some cases one needs not repeat a specific aspect over and over for every "instance" of the original program where we would like the specific action to take effect. To address this issue, we include a few simple mechanisms for aspect parameterization and naming akin to procedure definition and arguments. For instance, it is possible to indicate the application of a specific aspect (*loopTransf(var = j; factor=3)*) by invoking it in the *apply* section or by embedding it with the annotation *%@apply::loopTransf(var = j; factor=3)*. This invokes the aspect "*loopTransf*" with respect to variable "j" and with its "factor" parameter bound to the value 3. Unless otherwise stated in the argument list, all other details of the transformation remain as defined in the original definition of aspect "*loopTransf*". These include the location, which is for this particular transformation the entire loop construct and/or variables to be affected. This instantiation ability also requires that the definition of the aspect exists in the aspect code accompanying the MATLAB code or in a separate aspect repository.

The use of parameterized aspects and their instantiation may prove to be fundamental when considering reusable and higher-level aspects. Possibly, helping to structure in a very compact and easily maintained form a whole range of transformations. Parameterized aspects in turn will enable the definition of design-space-exploration strategies, e.g., by enabling the evaluation of transformations/specializations driven by different values.

## 2.7 Inner Aspects

Inner aspects are aspects that run for each intersection of the (outer) aspect that encloses them. This notion allows testing other intersection points that can use information defined by a specific intersection of the outer aspect. Figure 7 presents more complex examples based on the notion of inner aspects. These insert code in a function to print the number of iterations of each innermost loop with a pre-defined pattern. For each such loop, one needs to insert a statement responsible for the counting, a statement that initializes the counting variable to zero, and a statement that prints the value to the standard output. A generic and reusable way to do this is through inner aspects that are executed depending on the conditions of the enclosing aspect.

```
function r=f1(...)
   ...
   for j = 1:1:N1
     sum = sum + A(j);
   end
   ...
   for j = 1:1:N2
     A(j) = A(j)/sum;
   end
   …
end
```

(a) piece of MATLAB code.

```
aspect top
// locate innermost loops with a given pattern
select: { for <var> = 1:1:<const integer c1> <body b1> end } ::
position innermost, <b1> // use of the loop body joinpoint
identified by b1
apply: insert { <this.name+this.id> = <this.name+this.id> + 1; }
     :: execute before // before the loop body
 inner aspect a1
     select: {function *} // function header
     apply: insert {<super.name+super.id> = 0;}:: execute after
 end a1
 inner aspect a2
     select: {function ... <key k1> in {end}} :: position <k1>
     apply: insert {
           sprintf('loop executed %d', <super.name+ super.id>);
     } :: execute before
 end a2
end top
```

(b) inner aspects.

```
function r=f1(...)
  top_1 = 0;
  top_2 = 1;
  ...
  for j = 1:1:N1
    top_1 = top_1 + 1;
    sum = sum + A(j);
  end
  ...
  for j = 1:1:N2
    top_2 = top_2 + 1;
    A(j) = A(j)/sum;
  end
  …
  sprintf('loop executed %d', top_1);
  sprintf('loop executed %d', top_2);
end
```

(c) Code after weaving.

**Figure 7. The use of inner aspects.**

## 3. IMPLEMENTATION ISSUES

Figure 8 outlines the flow of our system implementation currently under development. Aspect modules, strategies, and MATLAB code are specified in separate source files. A front-end parses the input MATLAB code and converts the obtained abstract-syntax tree into a specific IR (intermediate representation). The tool used is TOM [3], a high-level program rewriting framework that can be used to manipulate/transform an intermediate representation of the input MATLAB program. TOM accepts the definition of the rules and the rewriting strategies [2] and includes a pattern matching engine. Tags embedded in MATLAB code to specify joinpoints (e.g., *%@here*) are processed and embedded in the adopted IR and passed in this form by the MATLAB compiler front-end to the other tools in the compilation flow.

Data types and shapes are made available as symbol tables to the tools in the compilation flow. A transformation engine plays the role of aspect weaver, receiving the IR as input and generating a modified IR that includes the features specified by the aspect modules. The weaver is being implemented using the paradigm of strategic programming as provided by TOM. It determines the sequences of aspects to execute based on the aspect strategies. Other concerns, such as monitoring and code transformations, are also composed to the IR of the original MATLAB program through the weaver, which yields a modified IR made available to the subsequent tools in the compilation flow. This modified IR can include, e.g., representations of additional code.

Code generators in the flow presented in Figure 8 include the MATLAB and C generators [8]. Each one is important for different facets of the approach. Generation of code also takes advantage of the TOM code rewriting capabilities. Ongoing work focuses on developing an optimized C generator from MATLAB descriptions. We intend the C generator to use certain aspects to produce more efficient code with respect, e.g., to memory usage or to execution time.

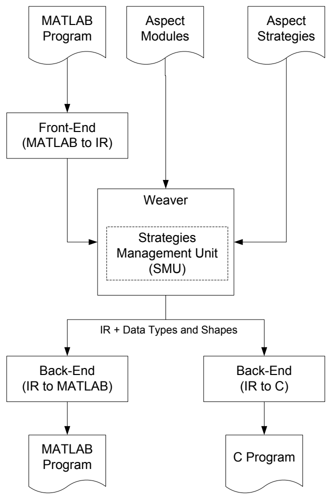

**Figure 8. Environment under development.**

## 4. RELATED WORK

Irwin et al. [6] describe AML, a system for sparse matrix computation that deals with crosscutting concerns such as execution time and data representation, using aspect-oriented programming principles [7]. AML allows programmers to write annotations regarding properties of sparse matrices separately from the main functionality. Readability and maintainability of the code is not adversely affected by non-functional aspects. The authors report that their AML codes have similar speed to a standard version, yet they are smaller and less complex. They propose an aspect, called "data representation" that is relevant for our work. This aspect defines five axes for representing data: element type, dimension, representation, ordering, and orientation.

Aslam et al. [1] describe AspectMatlab, a language inspired on AspectJ that extends MATLAB with aspect-oriented features. The authors focus on describing the static flow analysis techniques used to reduce dynamic checks required in the woven code for monitoring, and for tracking the sparsity of manipulated matrices. AspectMatlab is thus geared towards scientific codes using aspect modules that define *patterns-actions* that support constructs such as loops, loop bodies, array accesses, and function calls.

The primary difference between AspectMatlab and our approach is that we address aspect-oriented features not only able to help programmers to monitor runtime features, but also to evaluate different implementations of a single MATLAB specification. The central characteristics of our approach are powerful pattern-based advices and actions, and parameterization capabilities that improve aspect reuse. Both approaches need to deal with the future evolution of the base language, an issue that is particularly relevant in the case of proprietary languages such as MATLAB. Evolution can be more flexibly handled when keeping a strict separation between a MATLAB base and the aspects.

Our proposal differs from the above efforts as the proposed aspects aim to help developers in exploring different implementations of a given MATLAB specification without the need to change the original code for each candidate optimization, thus avoiding the need to manage multiple hand-written versions of the base code. We believe that the majority of the proposed aspects are not suitable to be embedded in the original specification as annotations. Firstly, that would make the code less legible and less maintainable. Secondly, that would still require multiple code versions when exploring different data types for a given variable. Thirdly, some rules are intended to be applied *globally*, not just to a specific function. In our approach, explorations can be performed with the same specifications by employing different aspect rules as we use a declarative type of aspect that can be applied both locally and globally.

## 5. CONCLUSION

This paper presents an approach for specifying transformations of MATLAB programs in an aspect-oriented style, with a focus on optimization concerns. We describe a set of features for a domain-specific language to program strategies, organized as aspect modules. Studies conducted so far suggest that the proposed features help developers to explore a number of concerns without "polluting" the original code and avoiding the need for multiple versions of the base program. Keeping a strict separation between a MATLAB base and new aspect-oriented features contributes to improved maintenance, readability, and reuse of both base programs and aspects. Work in progress includes studies about additional aspect-oriented features, development of the weaver, experiments on the implementation of the transformation engine, and implementation of the main concepts in our compiler framework.

## ACKNOWLEDGMENTS

This work has been partially supported by FCT (Portuguese Science Foundation) under grant (POCTI, PTDC/EIA/70271/2006).

## REFERENCES

[1] Aslam, T., Doherty, J., Dubrau, A., and Hendren, L. 2010. AspectMatlab: An Aspect-Oriented Scientific Programming Language, in Proc. of the Intl. Conf. on Aspect-Oriented Software Development (AOSD), March, 2010.

[2] Balland, E., Moreau, P.-E., and Reilles, A. 2008. Rewriting Strategies in Java, ENTCS vol. 219, Elsevier, pp. 97-111.

[3] Brauner, P., Kopetz, R., Moreau, P.-E., and Reilles, A. 2007 Tom: Piggybacking Rewriting on Java, RTA'07, LNCS 4533, Springer, pp. 36-47.

[4] Cardoso, J. M. P., Fernandes, J., and Monteiro, M. 2006. Adding Aspect-Oriented Features to MATLAB, in SPLAT!2006, at AOSD'06.

[5] Gradecki, J. D., and Lesiecki, N. 2003. *Mastering AspectJ: Aspect-Oriented Programming in Java*, Wiley.

[6] Irwin, J., Loingtier, J.-M., Gilbert, J., Kiczales, G., Lamping, J., Mendhekar, A., and Shpeisman, T. 1997. Aspect-Oriented Programming of Sparse Matrix Code, ISCOPE'97, Springer, pp. 249-256.

[7] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J. 1997. Aspect Oriented Programming, ECOOP'97.

[8] Nobre, R., Cardoso, J. M. P., and Diniz, P. C. 2010. *Leveraging Type Knowledge for Efficient MATLAB to C Translation*, Technical Report, Portugal, February 2010.

[9] The Mathworks Inc., http://www.mathworks.com