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
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 provides powerful ways to augment programs with information out of the scope of the base language while avoiding harming code readability and thus portability. MATLAB is a popular modeling/programming language that will strongly benefit of aspect-oriented programming features. For instance, MATLAB programmers could use aspects to provide information such as restrictions on allowed data types and/or values, monitoring specific aspects of the execution such as the effective dataset sizes or if a given variable ever assumes a specific value, without "polluting" the code with "check code". This paper describes the main concepts of a domain-specific aspect language (DSAL) for specifying transformations of MATLAB programs in view of supporting optimizations by facilitating the experimentation of alternative implementations. This DSAL specifies aspect modules structured in three sections: *intersections* equivalent to AspectJ poincuts, *actions* equivalent to AspectJ advice, and *conditions* that control triggering of actions. Support for aspect composition strategies and aspect parameterization of tokens from the base program are also supported. We believe the described features complement and enhance MATLAB programming in substantial and valuable ways.

Keywords

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

1. INTRODUCTION

MATLAB [1] is an interpreted, imperative programming language mainly based on matrix-shaped double precision data types and operations on them. It is widely used in scientific computing, control systems, signal processing, image processing, system engineering and simulation. MATLAB relies heavily on matrix data types and provides some base parametric primitive data types such as integer and fixed-point variables. 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. These unwarranted specializations are exacerbated by changing program requirements (e.g., power vs. performance) or 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 exploiting non-uniform fixed-point representations, monitoring certain variables during a timing window, or to include handlers to watch specific behaviors, the programmer is overwhelmed by cumbersome, error-prone and tedious tasks. Each time these kinds of features are necessary, invasive changes on the original code are required, as well as the insertion of new code related to non-core concerns. This problem is felt in other implementation issues as well, since MATLAB can be regarded as a specification rather than an implementation language.

In previous work [2], 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. Our current efforts focus on augmenting the MATLAB programming methodology by using a DSAL with more powerful aspect-oriented concepts. Those concepts will allow the exploration of specific features within the system's design and implementation space, debugging and monitoring, and specification of programmers's knowledge about an algorithm not directly captured in the MATLAB program structure. In our approach, a single version of the specification can be used throughout the entire development cycle rather than maintaining multiple versions, as is presently the case. We believe this separation helps the development, simulation, exploration and implementation phases.

In this paper we address some of the issues resulting from the inflexibility of existing programming languages, using an aspect-oriented approach. We propose aspect modules expressed in a domain-specific language based on the key concepts of joinpoint selection (select) composition (apply) and conditional binding

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

(when), through which programmers providing to a compiler/runtime system additional knowledge about program facets that are otherwise hard or impossible to derive from the original program.

A simple use case of the aspects supported relates to variable shapes 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 on specific input values or sizes of variables. We also propose an abstract strategy mechanism that enables programmers to explore the optimization space by applying a series of program transformations subject to values resulting from the specific aspect execution. For instance, one can derive a simple strategy that will 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 is free of language enhancements and sources remain legal MATLAB. The proposed DSAL enables programmers to retain the obvious advantages of a single source program representation while allowing the implementations to explore 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 of our approach. Section 3 briefly discusses implementation issues. Section 4 compares our approach to related work. Finally, we conclude in section 5.

2. ASPECT CONCEPTS AND DESIGN

Figure 1(a) illustrates the structure of an aspect module and its code sections, as the main component of the DSAL, and Figure 1(b) shows an example of an aspect. Each aspect module can have several *select-apply-when* 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 execute. Different sequences can be structured as strategies.

aspect <name> (**input**: ... **output**: ...)? (**select**: ... end **apply**: ... end (**when**: ... end)?)+ end <name>	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
(a) the structure of an aspect module.	(b) example of an aspect module.

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

Figure 2(b) shows the resulting code of applying the aspect in Figure 1(b) to the MATLAB code in Figure 2(a). A more generic aspect module is illustrated in Figure 3 and gives the same result as the aspect in Figure 1(b), if applied as *warning_too_big({sum,A}, 10000);* to the MATLAB code in Figure 2(a).

We now describe our approach concepts, which includes joinpoint selections, advice-like actions, conditions, and strategies.

... for j = 1:1:N sum = sum + A(j) * B(j+N); end outa(i) = sum; ...	... 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; ...
(a) piece of MATLAB code.	(b) MATLAB code with logging code (underlined and in italic).

Figure 2. Code inserted for logging if certain variables exceed a value.

```
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

Our focus is on maximizing configurability, which takes precedence over long-term maintainability. Thus, the proposed joinpoint model covers virtually any point in the code of a program. Unlike in many AOP approaches including AspectJ [3], joinpoints are not restricted to method invocations, object instantiations, and variable accesses. Joinpoints can be identified by a name related 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, one can use annotation-like tags embedded in MATLAB comments to specify joinpoints. This approach uses the convention that such tags must start with '%@', e.g., %@here1, %@loop1. The keyword "%" is the beginning of a comment line in MATLAB and consequently the resulting annotated MATLAB code remains legal MATLAB.

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.

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. Regarding the position at a particular joinpoint, the action is activated (i.e., if enabled by its trigger, the corresponding action is executed). We support the three usual types: "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 join-

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

(a) MATLAB base code.	
... for i=1:2:100 A(i) = B(i) + 1; A(i+1) = B(i+1) + 1; end ...	
(c) resulting code after weaving base and aspect	(b) aspect module with intersection pattern.

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., 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, with one or the other being selected depending on the evaluation of the condition.

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

(a) dynamic condition.	(b) example of code after weaving.

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

2.4 Aspect Strategies

As aspect components are declarative in nature, we allow programmers to specify a specific sequence for the application of aspects through a *strategy*. For example, the aspect strategy "*A: aspect1 → aspect2 → aspect3*" (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-like 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.

Aspect strategies define possible flows of aspects and are defined in *aspect management units* (see examples in Figure 6). For each call of an aspect, information can be returned to the aspect management unit. This returned information 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
	apply: B
	strategy B
	do
	a1=aspect1;
	while(a1.modified);
	end B

(a) strategy for a sequence of aspects.	(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 from Figure 6 illustrate strategies used by the aspect management unit. Example (a) illustrates an aspect strategy for defining a sequence of 3 aspects. Example (2) 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*). With this, base code can be modified/specialized assigning different values to variables present in the code, e.g, a segment of code <body> can use a variable defined as <var> outside the code in the <body> and the reference <var> can be used to modify the name of the variable referred by <var>, or to substitute the name of the variable referred by <var> with the same name concatenated to "+1" as illustrated in Figure 4. These variables have attributes that can be used in the action and condition sections of the aspects. Attributes are identified by the name of the variable followed by '.' and the attribute name (e.g., "a.name" for the variable <var a>).

One important feature of these variables is that they can be referred in actions that can modify other inner variables. The code *insert{p1(replace <c1.value> with "100")}* in which "*p1*" identifies a code pattern is an example. 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 can transpose different values for the same pattern based on the program location where that 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 aspect code or by embedding it with the annotation *%@apply::loopTransf(var = j; factor=3)*. This replaces the already defined "*loopTransf*" aspect with its "factor" parameter bound to the value 3. Unless otherwise stated in the argument list, all other aspects of the transformation remain as defined in the (possibly unique) 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 key when generating higher-level aspects, thus helping to structure in a very compact and easily maintained form a whole range of transformations. These in turn will enable the definition of design-space-exploration strategies.

As with any declarative mechanism, it is conceivable, although not desirable, that aspects give rise to conflicts. One example is to declare the type of a given variable as integer while a second aspect declares the range of values for the same variable to be in the real or floating-point domains. The compilation tool will execute the aspects being the final code the one after the sequence of aspects in the strategy.

2.7 Inner Aspects

Inner aspects are aspects that run for each intersection of the (outer) aspect that encloses them. This notion allows to test other intersection points that can use information defined by a specific intersection of the outer aspect. Figure 7 presents more elaborate 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.

3. IMPLEMENTATION ISSUES

Figure 7 outlines the system implementation. 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 [4], 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 rules and rewriting strategies [5] and includes a pattern matching engine.

```
1.    function r=f1(...)
2.      ...
3.      for j = 1:1:N1
4.        sum = sum + A(j);
5.      end
6.      ...
7.      for j = 1:1:N2
8.        A(j) = A(j)/sum;
9.      end
10.     …
11.   end
```

(a) piece of MATLAB code.

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

```
1.    function r=f1(...)
2.      top_1 = 0;
3.      top_2 = 1;
4.      ...
5.      for j = 1:1:N1
6.        top_1 = top_1 + 1;
7.        sum = sum + A(j);
8.      end
9.      ...
10.     for j = 1:1:N2
11.       top_2 = top_2 + 1;
12.       A(j) = A(j)/sum;
13.     end
14.     …
15.     sprintf('loop executed %d', top_1);
16.     sprintf('loop executed %d', top_2);
17.   end
```

(c) Code after weaving.

Figure 7. The use of inner aspects.

Tags embedded in MATLAB code to define specific 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 with the IR of the original MATLAB program

through the weaver, which yields a modified IR made available to the subsequent tools in the development process. This modified IR can include, e.g., representations of additional code.

Code generators in this flow include the MATLAB and C generators. Each is important for different aspects of the approach. Generation of code also takes advantage of the TOM [4] 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 (e.g., with respect to memory usage or to execution time).

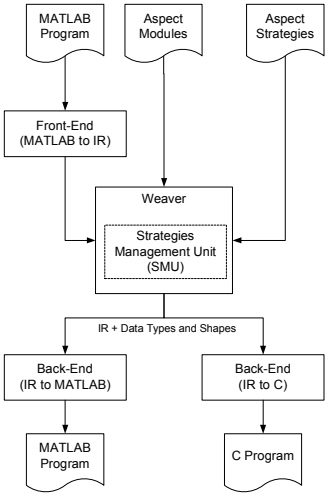

Figure 8. Environment under development.

4. RELATED WORK

In [6], Irwin et al. present 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 the programmer to write annotations that represent properties of sparse matrices separately from the main functionality. Thus, readability and maintainability of the behavioral code is not (negatively) affected by non-functional aspects. The AML system seems to have a satisfactory result, since the authors report that their code in AML has similar speed than a standard version, yet it is smaller and less complex. They propose an aspect, called "data representation" that is relevant for our work. This aspect defines 5 axes for representing data: element type, dimension, representation, ordering, and orientation.

Aslam et al [8] describe AspectMatlab, a new language that extends MATLAB with aspect-oriented features. The design of AspectMatlab is inspired on AspectJ, adapted to the specific features of MATLAB. The authors focus on describing the technical issues arising in the context of a weakly typed language and the static analysis techniques used to derive information needed for composing aspects on the remaining parts of the system without compromising performance of the generated system. The primary difference between AspectMatlab and the approach described here is that we maintain MATLAB sources separate from aspect-specific constructs, while AspectMatlab merges them into a single specification. While AspectMatlab offers a tighter integration between the "base" code and aspects, our approach was designed to minimize dependencies between the MATLAB original sources and aspects. Keeping aspects separate from plain MATLAB code provides additional guarantees of such independence. 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 as in the case of MATLAB. Evolution issues can be more flexibly handled when a strict separation between a MATLAB base and aspects is maintained.

Our proposal differs from these in that although type refinement may help compilers to produce better code, the aspects we propose are intended to help developers to model and to explore different implementations of a given MATLAB specification without the need to change the original code for each individual candidate optimization, thus avoiding the need to manage multiple versions of the base code. We also believe that most of the proposed aspects are unsuitable to be embedded in the original specification as annotations. First, that would make the code less legible and less maintainable. Second, this would still require multiple code versions when exploring different data types for a given variable. Third, some of the 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 local 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. From the studies conducted so far, the features proposed 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 MATLAB behavior and the new aspect-oriented features (e.g., data type assignments) 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 the implementation of the main concepts in our compiler framework.

REFERENCES

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

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

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

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

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

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

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

[8] T. Aslam, J. Doherty, A. Dubrau and L. Hendren, "AspectMatlab: An Aspect-Oriented Scientific Programming Language," Sable Tech. Report No. sable-2009-03, McGill University, 2009.