MATLAB ASPECTS

Project: AMADEUS

João M. P. Cardoso, Pedro C. Diniz, Miguel P. Monteiro, João M. Fernandes, João Saraiva

Technical Report

TR-AMADEUS-01-2009

November 2009

(with some revisions in Dec. 2009)

Contribution	Institution	Version	Date
João M. P. Cardoso	FEUP	0.1	July 2009
João M. P. Cardoso	FEUP	0.2	Oct. 2009
Pedro C. Diniz	IST/INESC-ID	0.3	Nov. 2009
João M. P. Cardoso	FEUP	0.4	Nov. 2009
João M. P. Cardoso and João M. Fernandes	FEUP	0.5[1]	Dec. 2009
Pedro C. Diniz and João M. P. Cardoso	FEUP and IST/INESC-ID	0.6 and 0.7	Dec. 2009
Miguel P. Monteiro	UNL	0.73	Dec. 2009

Authors and Affiliations:

João M. P. Cardoso
Departamento de 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. Engenharia Informática
Universidade Nova de Lisboa, Costa da Caparica
PORTUGAL
mmonteiro@di.fct.unl.pt

João M. Fernandes
Dep. Informática / CCTC
Universidade do Minho, Campus de Gualtar, Braga
PORTUGAL
jmf@di.uminho.pt

João Saraiva
Dep. Informática / CCTC
Universidade do Minho, Campus de Gualtar, Braga
PORTUGAL
jmf@di.uminho.pt

[1] Result of the meeting in Braga on December 4th, 2009 with João Saraiva, Miguel P. Monteiro, João M. Fernandes, and João M. P. Cardoso.

Table of Contents

Abstract

This report describes and outlines our main concepts regarding aspect-oriented concepts for MATLAB programming. The report begins by introducing the notion of aspects and how they can be used to provide a wide variety of applications such as profiling of the code execution by observing specific values and specific conditions of the execution. The proposed aspects consider code specialization (data types and computations), exploration primitives by injecting special code in order to explore certain aspects of the application (e.g., data types, word lengths, precision), tracking and dealing with special results and conditions, etc. We provide several examples of the use of aspects, including a more advanced example we call dynamic value injection where the programmer can perform testing of the MATLAB program by injecting specific values upon specific predicate conditions. This report also describes the framework of the AMADEUS project in which this aspect technology is currently being developed for MATLAB systems.

1 Aspect-Oriented Programming and MATLAB

Aspect-oriented programming (AOP) [1, 2] is an emerging paradigm characterized by a systematic approach to software modularity, with a focus on the modularization of crosscutting concerns. AOP was proposed as a way to tackle limitations and deficiencies in traditional paradigms caused by the "tyranny of the dominant decomposition" [3], i.e., paradigms that support only a single criterion to decompose systems into modules. Most concerns that do not align with the dominant decomposition cut across the boundaries of the modules of the system. The usual symptoms of the presence of such crosscutting concerns are code tangling and scattering [1].

An early, and widely cited, proposal of what comprises the essence of AOP rests on the concepts of quantification and obliviousness [4]. Quantification is the ability of an AOP language to specify a predicate that can match a variety of points in the static module definitions and dynamic object interaction graphs. Obliviousness is the ability of the modules of a system to be the subject of quantification without having to provide explicit hooks to expose the join points that the aspect modules intend to quantify over. Quantification can also be viewed as a "program-augmentation" approach where one decouples some concerns into separate aspect modules – henceforth referred simply as *aspects* – from the remaining modules of the program, by making aspects take the form of transformation specifications over the modules related to the primary functionality. This transformational approach to quantification, taken in this report, seems equally powerful in providing productivity tools (such as a compilers and aspect weavers), features that are typically impossible to express (or extremely hard to derive) when using traditional mechanisms.

The quantification capabilities under development in the context of AMADEUS include the definition of expected ranges of variable values and assertions that must hold at specific program points. While some of these aspect modules can be viewed as essentially (static)

annotation-based code injection and geared towards execution analysis, other features are also planned to inject values during runtime execution of the program, whenever certain conditions occur.

It has long been established that separation of concerns [5] contributes to improve code understandability and maintainability and may help tools (such as compilers) to improve productivity. To attain enhanced separation of concerns in MATLAB source code bases, we proposed in [6] an aspect-oriented language to enrich MATLAB with declarative aspect rules specifically tailored for the composition of crosscutting concerns such as monitoring; data type binding; function specialization and configurations; debugging, handling of abnormal conditions, including exceptions. As a continuation to that work, we focus here on more advanced concepts proposed within the AMADEUS project[2].

The main goal of the AMADEUS project is to augment MATLAB program specifications with advanced compilation approaches, namely by leveraging user's knowledge for the automatic development of code transformation strategies. In the AMADEUS approach, an input MATLAB program/model can be augmented with specifications to guide a given set of code transformations or/and to add information about certain properties of the input algorithm. Ongoing work focuses on the weaver responsible to perform code transformations taking as two types of inputs: (1) files with MATLAB code and (2) files containing aspect rules. In the end, the weaver will assist MATLAB and C code generators by outputing a representation of the MATLAB program much closer to the implementation (e.g., with shape and type information), enriched by users' knowledge about the algorithm, etc.

Figure 1 outlines the proposed environment. Aspect rules and MATLAB code are specified in separate source files. A transformation engine playing the role of aspect weaver generates modified MATLAB code that includes the features specified by the aspect rules. We also plan to develop an optimized C code generator from MATLAB descriptions. The C code generator may also use certain aspects to produce more efficient code (e.g., with respect to memory usage or to execution time).

Figure 2 presents more details about the front-end tool in the compilation environment. The original MATLAB code is translated to TIR (TOM[3] intermediate representation (IR)). The specification of crosscutting concerns is translated to an intermediate representation and the data types and shapes are made available as symbol tables to the subsequent tools in the flow. The other concerns (such as monitoring, code transformations) are then composed with the TIR of the original MATLAB program through a weaver yielding a modified TIR that is made available to the subsequent tools in the development process. This modified TIR can include, e.g., representations of additional code. The environment is capable to weave the input MATLAB code and to produce new/modified MATLAB code and/or to translate this MATLAB code into C code.

[2] AMADEUS: ASPECTS AND COMPILER OPTIMIZATIONS FOR MATLAB SYSTEM DEVELOPMENT, research project partially funded by the portuguese science foundation (FCT): POCTI, PTDC/EIA/70271/2006.
[3] http://tom.loria.fr/

For the core part of the tools we use TOM [7], a high-level program rewriting framework that can be used to manipulate/transform an intermediate representation (TOM-IR, a representation of a DAG is used) of the input MATLAB program. TOM permits rules and rewriting strategies to be defined [8] and includes a pattern matching engine.

The code generators proposed in AMADEUS include the MATLAB and the C code generators. Each one is important for different aspects of the approach. The generation of code also takes advantage of the TOM [7] code rewriting capabilities.

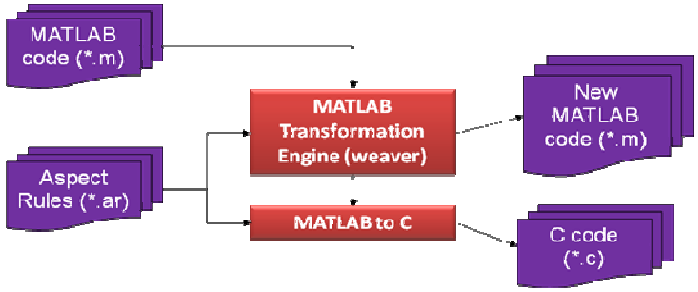

Figure 1. Outline of the MATLAB-based system augmented with aspect rules (rectangular boxes represent the two main concepts in the environment: the MATLAB weaving to produce MATLAB and C code).

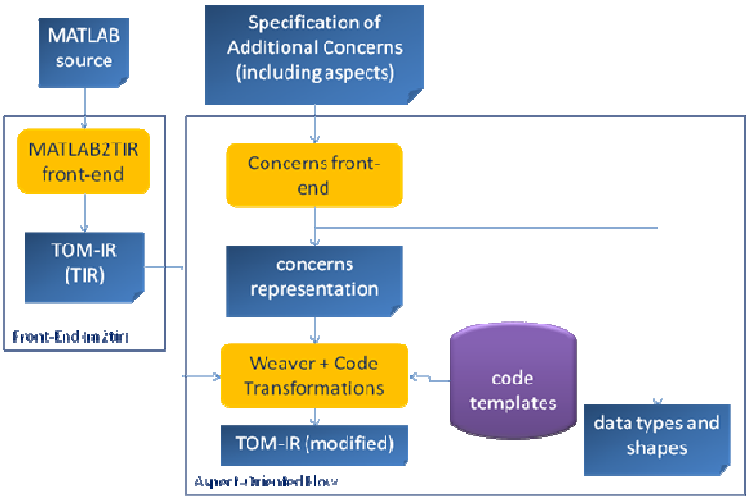

Figure 2. Front ends of the AMADEUS approach.

In this report we focus on the description of a set of simple static aspects used for code generation and transformations geared towards execution analysis. Specifically, we focus on aspects that augment a working MATLAB program with the ability to specify program characteristics such as data types and shapes of variables, data dependencies, optional functional implementations, parameterization capabilities that are either beyond the expressive scope of MATLAB or very hard to derive via static compiler analysis.

In our approach, we consider the following AOP-specific mechanisms:

- **Pointcut-like intersections** that identify specific points in the program (joinpoints) where a given aspect is to be applied or observed, i.e., program locations in the code where an aspect composes with program code. Typical intersections are: all uses of a set of

variables, a particular definition of a variable, an invocation of a particular method, and a certain point in the code.

- **Advices/Actions** are related to the action that is required at a particular joinpoint or at a set of joinpoints. The actions may include monitoring of variable properties and behavior (values, shapes), transforming code, inserting/replacing/removing code, transforming data-types, or performing design space exploration.
- **Conditions** are enablers or disablers of the trigger of a particular action. These conditions are optional and can be "static" or "dynamic". Static conditions refer to conditions that are checked before weaving (or static transformation of the code). Dynamic conditions refer to conditions that are checked during runtime.

Each aspect can be considered as a rule that can include intersections, advices and conditions. According to this approach, aspects and their rules are defined in source files separated from the MATLAB code. The rules are defined declaratively as initially proposed in [6]. To impose a certain sequence of rules the user needs to specify strategies as is shown in section 2.

In the following sections, we describe in more detail some of the concepts being used in the AMADEUS approach and illustrate the use and effects of several "static" aspect specifications. The examples are organized in sections according to the considered concern.

2 Concepts of the AMADEUS approach

The approach presented in this report is based on an aspect-oriented domain-specific language. Figure 3(a) illustrates the aspect and its code sections[4] as the main component of the language and Figure 3(b) shows an example of an aspect. Each aspect can have several intersection-advice-conditions sections and all must be considered when executing that aspect. Aspects may have input arguments and return output information. Inputs and outputs may include parameters to specialize an aspect, joinpoints to limit the scope of the intersection of an aspect to a set of intersections previously determined by another aspect, variables, etc. The sequence of aspects to be run is specified by the user. Different sequences are considered strategies.

We describe next in detail the concepts of our approach, which includes join points, advices, conditions, and strategies.

[4] An implementation possibly more understandable for MATLAB programers can use select/apply/when(for identifying the three sections: intersection/advice/conditions.

aspect <name> input: ... output: ... (intersection: ... advice: ... (conditions: ...)?)* end	 aspect rule1 input: var_name, const_value intersection: all uses <var a1> in {<var_name>} advice: insert {*if <a1.name> >= <const_value> warning ('<a1.name> too big! %f',<a1.name>); end*}:: execute before end
(a) an aspect and its sections.	(b) example of an aspect.

Figure 3. Aspect module, the main component of the language.

2.1 Classification of Aspects

We classify aspects in two orthogonal dimensions as being either static or dynamic and being either declarative or transformative. Static aspects convey information or direct the application of code transformations that are performed at compilation time and are thus confined to syntactic transformations or simple declarative aspects. Declarative aspects can include static information regarding the types and shape of a variable or dynamic information regarding the range of assumed values.

Transformative aspects can also be either static or dynamic. Static examples include the unrolling of a loop for a specific factor or to replace the references to variable "i" by a newly introduced variable, say "k". A dynamic transformative aspect reflects a transformation that depends on run-time data values and can be extensively used in design-space-exploration strategies. For instance, it might be desirable to change the precision requirements of a specific variable at run-time in response to the accumulated data values or even change the unrolling of a specific loop based on the target architecture to look for an optimal unrolling factor. Figure 4 depicts the overall classification of aspects.

Aspects	*Static*	*Dynamic*
Declarative	types and shapes	range of values and types
Transformative	code insertion removal / modification	code replication and/or instantiation for performance tuning

Figure 4. Classification of Aspect in two dimensions: declarative vs. transformative and static vs. dynamic.

The code transformations involved in the transformative dynamic aspects require some specific support from the code generation and corresponding run-time execution environments. When generating multiple code versions corresponding for instance to different unrolling factors, the generated code may contain replicates of the loops (properly unrolled)

controlled by a “switch” statement that dispatches to the appropriate code variant. In addition, one can also include the generation of code instrumentation to monitor specific execution metrics such as time, cache hit rates and so forth.

Alternatively, a dynamic code generation strategy can be used where the tool will use the aspect to create an executable file that at run-time will generate and possibly recompile the code. All these code generation aspects must be adequately supported by the target infrastructure, which in many cases will disallow (for performance considerations) the use of dynamic code generation and compilation techniques.

2.2 Join Points

The joinpoint model from our approach covers any point in the code of a program. Unlike in many AOP approaches including AspectJ [9], 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) or a broader characteristic (all variables, all variables of type float, all invocations of a function). In addition, one can use tags embedded in MATLAB comments to identify joinpoints. The AMADEUS approach uses the convention that such tags must start with ‘%@’, e.g., %@here1, %@loop1. These tags are processed and are embedded in the adopted IR and passed in this form by the MATLAB front-end to the other tools of the compilation flow. Choice of the keyword “%” assumes that it is the beginning of a valid comment line and thus the resulting annotated MATLAB code is as valid an instance of MATLAB as the original unannotated version (i.e., the annotations are treated as plain comments by typical MATLAB parsers).

Figure 5 illustrates the use of the tags and how they are represented in the IR generated by the MATLAB front-end [10]. As illustrated in Figure 5, the IR also keeps the line numbers.

1. function [y] = f1(x1,x2) 2. 3. % this is a simple example to show the tags 4. 5. %@l1 6. y=x1*x2; %@l2 7. %@l3	Start(FunctionMFile(ConcIdentifier(Identifier("y",1)), Identifier("f1",1), ConcIdentifier(Identifier("x1",1), Identifier("x2",1)), ConcStatement(Statement(Pragma("%@l1",5),-16), Statement(Assign(Expression(Id(Identifier("y",6))), Times(ConcExpression(Id(Identifier("x1",6)), Id(Identifier("x2",6))),6,0,ConcDim()),6),11), Statement(Pragma("%@l2",6),-16), Statement(Pragma("%@l3",7),49)),1), 1)
(a) MATLAB example	(b) IR code for the MATLAB example

Figure 5. MATLAB example and the IR.

We use a number of keywords to position the intersection points in the code. Table I presents those keywords and describes their meaning.

Table I. Keywords to specify positions.

keyword	Description	Example (underline represents the final intersection)
innermost	Specifies innermost nested levels in blocks of code (e.g., nested loops)	... for i=1:10 ... for j=1:20 ... end ... end ... pattern with loop: innermost
innermost(<integer>)	<integer> is a positive integer number used to identify the level of the intersection in the nested levels counting from the innermost level	... for i=1:10 ... for j=1:20 ... end ... end ... pattern with loop: innermost(2)
outermost	Specifies outermost nested levels in blocks of code (e.g., nested loops)	... for i=1:10 ... for j=1:20 ... end ... end ... pattern with loop: outermost
outermost(<integer>)	<integer> is a positive integer number used to identify the level of the intersection in the nested levels counting from the outermost level	... for i=1:10 ... for j=1:20 ... end ... end ... pattern with loop: outermost (2)
leftmost	Specifies the leftmost use of an identifier or variable	a=b*c+b*d+b; <var> = b.leftmost
leftmost(<integer>)	<integer> represents the number of the use from the leftmost	a=b*c+b*d+b; <var> = b.leftmost(2)
rightmost	Specifies the rightmost use of an identifier or variable	a=b*c+b*d+b; <var> = b.rightmost
rightmost(<integer>)	<integer> represents the number of the use from the rightmost	a=b*c+b*d+b; <var> = b.rightmost(3)
All	consider all nested levels	... for i=1:10 ... for j=1:20 ... end ... end ... pattern with loop: all

More examples for defining intersections are presented in “Appendix B: Intersections, page 36”. Note, however, that the goal is to have an initial proposal for a set of joinpoint designators and not a final, finished joinpoint model. We expect to design a language and to go through a maturing process.

More elaborated intersections include a scheme to define intersection patterns by allowing lexical matching and approximate syntactic matching. Figure 6 and Figure 7 show examples of a pattern matching specification of an intersection. The two examples differ in that one uses a static condition (Figure 6) and the other a dynamic condition (Figure 7).

Intersection: { for <var a1> = 1:1: <const integer c1> <body> end } :: position innermost	for i=1:1:100 A(i) = B(i) + 1; end
(a) specification of a pattern based intersection.	(b) example of code with successful pattern matching.
Advice/Action: insert {for a1 = 1:2: c1 <body> <body(replace <a1.name> with <a1.name>."+1")> end} :: position around	Conditions: static { if <c1.value> % 2 == 0 }
(c) specification of an advice.	(d) static condition.
... for i=1:1:50 A(i) = B(i) + 1; A(i+1) = B(i+1) + 1; end ...	
(e) example of code after weaving.	

Figure 6. Example of intersection mechanism, using pattern matching and a static condition.

Intersection: - {for <var a1> = 1:1:<var a2> <body> end} :: position innermost	... for i=1:1:N A(i) = B(i) + 1; end ...
(a) specification of a pattern based intersection.	(b) example of code with successful pattern matching.
Advice/Action: insert {for <a1.name> = 1:2:<a2.name> <body> <body(replace <a1.name> with <a1.name>."+1")> end} :: position around	Conditions: dynamic {if <a2.name> % 2 == 0}
(c) specification of an advice.	(d) 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 % original code if pattern is unmatched for i=1:1:N A(i) = B(i) + 1; end end ...	
(e) example of code after weaving.	

Figure 7. Example of intersection mechanism, using pattern matching and a dynamic condition.

2.3 Advices/Actions

The actions to be performed play the same role as advices in AspectJ. The actions are associated with one or more join points. This approach uses three main actions: insert, replace, and remove; with the obvious meanings. With respect to the position at a particular joinpoint, the advice is activated (i.e. if enabled by its trigger, the corresponding action is executed). We distinguish three types: "around"[5] (over a joinpoint, i.e., the advice replaces the code in that joinpoint), "before" (the advice is executed before the code in that joinpoint), and "after" (the advice is executed after the code in that joinpoint). Recall that this joint-point can be either a high-level construct or a single occurrence of a variable identifier. Table II presents simple examples of the use of the advice position features.

Figure 6(c) shows an example of an advice for the intersection illustrated in Figure 6(a). This advice unrolls every loop that matches the specified pattern twice.

Table II. Keywords to define positions relative to the intersections points to run the advice.

keyword	Description	Example
after	run advice after join point	a=b*c; %@here1 @here1.insert{%comment}:: execute after a=b*c; %@here1 %comment
before	run advice before join point	a=b*c; %@here1 @here1.insert{%comment}:: execute before %comment a=b*c; %@here1
around	run advice over join point	a=b*c; %@here1 @here1.insert{%comment}:: execute around %comment

2.4 Conditions

Conditions are the enablers or disablers of the execution of an advice. An advice without conditions is always executed. Figure 6(e) and Figure 7(e) present examples of conditions: a static condition and a dynamic condition. In each case, the condition evaluates if the upper bound of the iteration interval is a multiple of 2. In the static condition, the advice (i.e., the code transformation) is executed only if this condition evaluates to true. The dynamic condition forces the weaver to include in the output code the original intersected code and the modified code according to the advice, being the execution of one or the other based on the evaluation of the condition.

[5] The words "instead" and "surround" seem to better describe the aspect effect, when compared with "around". Anyway, we adopt "around" as it is used in most AOP approaches such as AspectJ.

2.5 Strategies

As aspect rules are declarative in nature, we allow users to specify a specific sequence for the application of rules by a mechanism called "strategy". For example, the strategy "A: rule1 → rule2 → rule3" implies that the weaver must first perform rule 1, then rule2, and finally rule3. Each rule in the sequence may modify code and new modifications may follow previous modifications. Different sequences (strategies) may produce different results. Although finding the appropriate and correct strategy is an interesting research topic, in this work we mainly focus on the programming support for strategies.

We follow an imperative approach for specifying strategies. This approach also has mechanisms to perform typical programming languages control flow. This strategic programming must deal with the following issues:

- a rule can be applied recursively while a certain condition holds[6],
- execution of different rule sequences in paths enabled by conditions,
- the use of loops to repeat sequences of rules, and
- passing data between aspects.

Strategies define possible flows of aspects and are defined in aspect management units (as represented by the examples in Figure 8). For each call of an aspect, a list of sets of attributes can be returned to the aspect management unit. This list may consist of a set of aspect attributes for each intersection of the aspect in a given call.

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 of occurred in a previous aspect (e.g., a1=aspect1 → aspect2(a1.intersection_region)).

Table III. Attributes for aspect components.

Aspect attribute	Description
name	name of the aspect
id	a number identifying the intersection
modified	true if the aspect modified the code
status	...
intersected	true | false
number of intersections	...
intersection_region	

Figure 8 presents two examples of an aspect management unit. The first example illustrates a strategy defining a sequence of 3 aspects. The second example illustrates a strategy where an aspect is repeated until a certain condition does not hold.

[6] An aspect used to unroll a certain kind of loops (based on a pattern) can be called recursively in the nested loop structure until no modification occurs to perform loop unrolling over all its loops.

apply: strategy1 strategy1 aspect1 -> aspect2 -> aspect3 end	apply: strategy1 strategy1 do a1=aspect1 while(a1.modified); end
(a) a sequence of aspects	(b) an aspect repeated called while a condition holds

Figure 8. Examples of strategies in the aspect management unit.

2.6 Lifetime of Aspect Rules

In our approach we consider aspect rules with lifetimes that span over the entire compilation flow (from the input program to the generated output) or only in the first compilation stages. Aspect rules related to the insertion of MATLAB code span only until the insertion of code is done. However, aspect rules related to insertion of C code are delegated to the generator of C code. As expected, aspect rules need to be active until the specific stages of the compilation flow where they can be required or used. Although out of the scope of our current work, there may be aspect rules that remain active during the entire lifetime of a given application and not simply at compilation time. This is the case of the aspects that are dynamic in nature and need to live beyond the compilation phase of a program.

2.7 Variables for Referring and Modifying Code

In the intersection subsection of the aspect we can define some variables that can be used in the other two sections (advice and conditions). The current types of variables are presented in Table II where we show for each variable (first column) its attributes (second column) and a brief description (third column) . Table IV presents some examples of using those variables (in this case we illustrate for each variable in the first column, an example in the second column).

Code can be modified/specialized assigning different values to variables present in the code. For example, 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 6.

These variables have attributes (represented in Table II) that can be used in the advice 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 we can refer them 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 of that fact. In this case, the code related to pattern “p1” is inserted in a pointcut and the constant identified by “c1” in the pattern is replaced by “100”.

Variables can be also a mechanism to manage differences in the actions performed by the same aspect. For instance, they can transpose different values for the same pattern based on the program location where that pattern intersects.

Table IV. Proposed variables.

variables	attributes	Description
<var [name]>	name type shape size length maxvalue minvalue wordlength	variable used to refer to variables in the code
<stmt [name]> [+|*|?|{n}|{n,m}|{n,}]	content	variable to refer code statements
<body [name]>	content	variable to refer the statements in a body of a loop, function, or if-then structure.
<pattern [name]>	content	variable to refer a given pattern
<const [integer|real] [name]>	value	variable to refer literals (constants)
<ident [name]>	name type	variable to refer an identifier (it can be a variable or a function)
<key [name]>	name	variable to identify keywords
<tag [name]>	name	variable to identify tags

Table V. Examples using the variables proposed.

variables	Example
<var [name]>	Intersection: - <pattern p1> = {<var a1> "=" <const integer c1> ";"} :: position all Advice: - insert {p1(replace <a1.name> with "a1_const"}::execute around
<stmt [name]> [+|*|?|{n}|{n,m}|{n,}]	Intersection: - <pattern p1> = {"{" <stmt s1>+ "}"} :: position all Advice: - insert {p1(copy <s1.content>)}::execute after
<body [name]>	Intersection: - <pattern p1> = {"{" <body b1> "}"} :: position all Advice: - insert {p1(copy <b1.content>)}::execute after
<pattern [name]> <const [integer|real] [name]>	Intersection: - <pattern p1> = {if <var a1> == <const integer c1> <body> end} :: position innermost Advice: - insert {p1(replace <c1.name> with "100")}::execute around
<ident [name]>	Intersection: - <pattern p1> = {<var> "=" <ident f1> "(" <var>{3} ")"} :: position all Advice: - insert {p1(replace <f1.name> with "stub1")}::execute around
<key [name]>	Intersection: - <pattern p1> = {<key a1> 1:10 ";"} :: position all Advice: - insert {p1(replace <a1.name> with "parfor"}::execute around
<tag [name]>	Intersection: - <pattern p1> = {for <statement> <tag a1>} :: position all Advice: - insert {p1(replace <a1.name> with "@newtag"}::execute around

2.8 Generalization of Aspects

Generalization of the aspects are possible 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 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 calling it in the aspect code or by embedding it with the annotation %@apply::loopTransf(var = j; factor=3). This replaces the already defined aspect named "loopTransf" 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 the "loopTransf" aspect. 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 might prove to 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 declarative aspects give rise to conflicts. For instance, declaring the type of a given variables to be integer whereas in a second aspect declare the range of values for the same variable to be in the real or floating point domains. In these scenarios de compilation tool will choose the later (in the syntactic sense) aspect.

2.9 Inner Aspects

Inner aspects are aspects that run for each intersection of the aspect (outer) that encapsulates them. This notion is very important, because it permits to test other intersection points that might use information defined by a specific intersection of the outer aspect.

Figure 9 presents aspects responsible to insert code for counting and reporting the number of iterations of the loops identified with tags. Figure 10 presents a more sophisticated example, which uses the notion of inner aspects. In the example of Figure 10, one wants to print the number of iterations of each innermost loop with a pre-defined pattern in a function. 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 in the standard output (as has been exemplified in Figure 9 with specific aspects). A generic and reusable scheme to do this is to use inner aspects. Inner aspects may be executed based on the conditions of the aspect where they pretence.

1. function a=f1(...) %@begin 2. ... 3. for j = 1:1:N %@loop1 4. sum = sum + A(j); 5. end 6. ... 7. for j = 1:1:N %@loop2 8. A(j) = A(j)/sum; 9. end 10. … 11. end %@end	Intersection: - {@loop1} Advice: - insert {*cnt1 = cnt1 + 1;*}:: execute after Intersection: - {@loop1} Advice: - insert {*cnt2 = cnt2 + 1;*}:: execute after Intersection: - {@begin} Advice: - insert {*cnt1 = 0; cnt2=0;*}:: execute after Intersection: // last end in the function - {@end} Advice: - insert {*sprintf('loop executed %d', cnt1); sprintf('loop executed %d', cnt2;*}:: execute before
(a) piece of MATLAB code with tags.	(b) aspects.
1. function a=f1(...) *2. cnt1 = 0; cnt2 = 1;* 3. ... 4. for j = 1:1:N *5. cnt1 = cnt1 + 1;* 6. sum = sum + A(j); 7. end 8. ... 9. for j = 1:1:N *10. cnt2 = cnt2 + 1;* 11. A(j) = A(j)/sum; 12. end 13. … 14. *sprintf('loop executed %d', cnt1);* 15. *sprintf('loop executed %d', cnt2);* 16. end	
(c) Code after weaving.	

Figure 9. The use of tags.

12. function a=f1(...) 13. ... 14. for j = 1:1:N 15. sum = sum + A(j); 16. end 17. ... 18. for j = 1:1:N 19. A(j) = A(j)/sum; 20. end 21. … 22. end	aspect top() Intersection: // locate innermost loops with a given pattern - {for <var> = 1:1:<const integer c1> <body b1> end} :: position innermost, <b1> // use of the loop body joinpoint identified by b1 Advice: - insert {*<this.name+this.id> = <this.name+this.id> + 1;*}:: execute before // before the loop body inner aspect a1() Intersection: - {function ...} // function header Advice: - insert {*<super.name+super.id> = 0;*}:: execute after end a1 inner aspect a2() Intersection: // last end in the function - {function ... <key k1> in {end}} :: position <k1> Advice: - insert {*sprintf('loop executed %d', <super.name+super.id>);*}:: execute before end a2 end top
(a) piece of MATLAB code.	(b) inner aspects.
17. function a=f1(...) *18. top_1 = 0;* *19. top_2 = 1;* 20. ... 21. for j = 1:1:N *22. top_1 = top_1 + 1;* 23. sum = sum + A(j); 24. end 25. ... 26. for j = 1:1:N *27. top_2 = top_2 + 1;* 28. A(j) = A(j)/sum; 29. end 30. … 31. *sprintf('loop executed %d', top_1);* 32. *sprintf('loop executed %d', top_2);* 33. end	
(c) Code after weaving.	

Figure 10. The use of inner aspects.

3 Examples Using Aspects

We show herein examples of the aspects being proposed in the AMADEUS project.

3.1 Code Transformations: Insertion, Deletion and Modification

We now illustrate the use of aspect actions to remove a specific code section. In Figure 11 we present an aspect that will take effect at the MATLAB code line labeled as "@here1". Since an around execution is selected for this case and no code is included within the bracket, the code in the line of the tag is effectively removed. The example in Figure 12 is similar but allows for the removal of a section of code, rather than a single instruction.

<table>
<tr><td>…
23. for j = 1:1:N
24. sum = sum + A(j);
25. end
26. outa(i) = sum; %@here1
…</td><td>Intersection:
{@here1}
Advice:
insert {}:: execute around</td></tr>
<tr><td>(a) piece of MATLAB code with join-point identified by tag "here1".</td><td>(b) aspect with code between {}.</td></tr>
<tr><td colspan="2">…
for j = 1:1:N
sum = sum + A(j);
end
…</td></tr>
<tr><td colspan="2">(c) Code after elimination of line 4 in (a).</td></tr>
</table>

Figure 11. Elimination of one line of code.

<table>
<tr><td>…
1. %@begin1
2. for j = 1:1:N
3. sum = sum + A(j);
4. end
5. %@end1
6. outa(i) = sum;
…</td><td>Intersection:
{@begin1-@end1}
Advice:
insert {}:: execute around</td></tr>
<tr><td>(a) piece of MATLAB code with joinpoint identified by tags "begin1" "end1".</td><td>(b) aspect with code between {}.</td></tr>
<tr><td colspan="2">…
%@begin1
%@end1
outa(i) = sum;
…</td></tr>
<tr><td colspan="2">(c) Code after elimination of the segment of code from line 2 to line 5 in (a).</td></tr>
</table>

Figure 12. Elimination of a segment of code.

The next example in Figure 13 describes a replacement aspect. Here at the instruction specified as the @here1 tag we include an identifier, in this case "outa". In the action, we specify that we should insert the identifier "sumA" in effect replacing "outA" with "sumA".

… 1. for j = 1:1:N 2. sum = sum + A(j); 3. end 4. outa(i) = sum; %@here1 …	Intersection: - {@here1}: <var a1> in {outa} // variable outa in line identified by @here1 Advice: - replace <a1.name> with "sumA"
(a) piece of MATLAB code with joinpoint identified by tag "here1".	(b) aspect.
… for j = 1:1:N sum = sum + A(j); end sumA(i) = sum; %@here1 …	
(c) MATLAB code after weaving.	

Figure 13. Code modification (the name of a variable in this example).

The insertion of code also permits to verify certain conditions and to compute according to those conditions. Figure 14 shows a simple example where a division by zero needs to be identified in runtime and, whenever it occurs, the calculations in line 2 of the example (Figure 14(a)) are replaced by the assignment of 0 to a(i). Note in this case that we assume *func* has no side effects and we do not need to execute it for each iteration of the loop to maintain the values of a(i) for the other iterations (i.e., for the iterations where b(i) does not equal zero). With a similar mechanism we may define traditional – or even more advanced try-catch – constructs.

… 1. for i = 1:n 2. a(i) = func(i)/b(i); %@here1 3. end …	Intersection: - {@here1} Advice: - insert {if b(i) == 0 a(i) = 0; else}:: execute before - insert {end}:: execute after
(a) piece of MATLAB code with joinpoint identified by tag "@here1".	(b) aspect.
… 1. for i = 1:n 2. if b(i) == 0 a(i) = 0; else 3. a(i) = func(i)/b(i); %@here1 4. end 5. end …	
(c) MATLAB code after weaving.	

Figure 14. Injection of code to avoid a division by zero exception and to continue execution with pre-defined results (zero is used in this case, but one can use the highest value for a(i) for each b(i) equal to zero).

3.2 Execution Monitoring: Source-level Instrumentation

The next example in Figure 15 illustrates an aspect used for source-level instrumentation for monitoring the specific value of a loop accumulation variable. In this specific case, the transformed code includes a simple threshold test at each loop iteration. The locus of

applicability is every instruction in the segment of code between @begin and @end where variable "sum" is used (read in this case). Without the use of "{@begin-@end}", the intersections would have applied over the entire MATLAB function (as shown in Figure 16).

… sum = 0; … %@begin for j = 1:1:N sum = sum + A(j) * B(j+N); end outa(i) = sum; %@end …	Intersection: - {@begin-@end}: all uses <var a1> in {sum} Advice: - insert { *if <a1.name> >= 10000 warning ('<a1.name> too big! %f', <a1.name>); end* }:: execute before
(a) piece of MATLAB code.	(b) aspect with MATLAB code between {}.
… for j = 1:1:N *if sum >= 10000 warning ('sum too big! %f',sum); end* sum = sum + A(j) * B(j+N); end *if sum >= 10000 warning ('sum too big! %f',sum); end* outa(i) = sum; …	
(c) MATLAB code after weaving. Code inserted is underlined and in italic.	

Figure 15. Insertion of code to report the value of a single variable before every use in a segment of code and if its value is greater than a certain value.

… sum = 0; … for j = 1:1:N sum = sum + A(j) * B(j+N); end outa(i) = sum; …	Intersection: - all uses <var a1> in {sum} Advice: - insert { *if <a1.name> >= 10000 warning ('<a1.name> too big! %f', <a1.name>); end* }:: execute before
(a) piece of MATLAB code.	(b) aspect with MATLAB code between {}.
… *if sum >= 10000 warning ('sum too big! %f',sum); end* sum = 0; … for j = 1:1:N *if sum >= 10000 warning ('sum too big! %f',sum); end* sum = sum + A(j) * B(j+N); end *if sum >= 10000 warning ('sum too big! %f',sum); end* outa(i) = sum; …	
(c) MATLAB code after weaving. Code inserted is underlined and in italic.	

Figure 16. Insertion of code to report the value of a single variable before every use, whenever its value is greater than a certain constant.

Figure 17 illustrates an example using two variables instead of one. The approach simply requires a generic advice to insert one message per variable. The use of the expression <var> permits to change the name of the variable ("sum" or "B") in the message to output according

to the variable name in each intersection. Note that the attribute name of <var> includes the name of the variable ("sum" and "B") and its access pattern ("B(j+N)" for the variable "B").

… for j = 1:1:N sum = sum + A(j) * B(j+N); end outa(i) = sum; …	Intersection: - all uses <var a1> in {sum, B} Advice: - insert { *if <a1.name> >= 10000 warning ('<a1.name> too big! %f',<a1.name>); end*}:: execute before
(a) piece of MATLAB code.	(b) aspect using a template based approach.
… for j = 1:1:N *if sum >= 10000 warning ('sum too big! %f',sum); end* *if* B(j+N) *>= 10000 warning (*'B(j+N) *too big! %f',* B(j+N)*); end* sum = sum + A(j) * B(j+N); end *if sum >= 10000 warning ('sum too big! %f',sum); end* outa(i) = sum; …	
(c) MATLAB code after weaving.	

Figure 17. Insertion of code to report the value of two variables before every use and if their value is greater than a certain value. The code inserted is based on a template based approach.

Figure 18 illustrates a simple case where the insertion of the code occurs just immediately before the line identified by the tag "@here1". Figure 19 illustrates another example of an aspect using code insertion in positions identified by tags. The code inserted in the example from Figure 19 is responsible for dynamically checking if the index values to the matrix are valid. An alternative aspect producing the same result but using only one intersection-advice rule is represented in Figure 20.

… for j = 1:1:N sum = sum + A(j) * B(j+N); %@here1 end outa(i) = sum; …	Intersection: - {@here1} Advice: - insert {*sprintf('sum = %f', sum);*}:: position before
(a) piece of MATLAB code. The tag "here1" refers to the line.	(b) aspect.
… for j = 1:1:N *sprintf('sum = %f' ,sum);* sum = sum + A(j) * B(j+N); end %@here1 outa(i) = sum; …	
(c) MATLAB code after weaving.	

Figure 18. Code insertion in a specific location identified with a tag.

<table>
<tr><td>…
%@ loop1-begin
for j = lower:1:upper
sum = sum + A(j);
end
%@ loop1-end
…</td><td>Intersection:
- {@loop1-begin}
Advice:
- insert {if lower >= 1 && upper <= length(A)}:: position before

Intersection:
- {@loop1-end}
Advice:
- insert {else warning('array limits exceeded!'); end}:: position after</td></tr>
<tr><td>(a) piece of MATLAB code.</td><td>(b) Aspects.</td></tr>
<tr><td colspan="2">…
<u>if lower >= 1 && upper <= length(A)</u>
%@loop1-begin
for j = lower:1:upper
sum = sum + A(j);
end
%@loop1-end
<u>else warning('array limits exceeded!'); end</u>
…</td></tr>
<tr><td colspan="2">(c) MATLAB code after weaving.</td></tr>
</table>

Figure 19. Aspect to check array limits (this example does not use the pattern matching and it is specific to the code shown).

Intersection:
- {@loop1-begin-@loop1-end}

Advice:
- insert {if lower >= 1 && upper <= length(A)}:: position before
- insert {else warning('array limits exceeded!'); end}:: position after

Figure 20. Alternative aspect using a code segment joinpoint.

3.3 More Advanced Code Transformations

Loop transformations are one of the most important compiler optimizations when considering execution time acceleration and/or energy consumption improvements. One of the most well-known transformation is loop unrolling (full or partial). Figure 21 and Figure 22 show examples of aspects to instruct the weaver to perform partial loop unrolling of the loop by a factor of two if the iteration space is multiple of 2. The example in Figure 22 instructs the weaver with high-level primitives.

The same transformation can be applied with the use of pattern-matching as in the example shown in Figure 6. To apply it to a specific loop and not to all of the loops that matches that pattern, we may restrict the scope of the intersection to the code region that includes the loop as illustrated in Figure 23.

… for j = 1:1:100 %@loop1 sum = sum + A(j); %@line1 end …	Intersection: - {@loop1} Advice: - insert {for j = 1:2:100}:: position around Intersection: - {@line1} Advice: - insert {sum = sum + A(j+1);}::position after
(a) piece of MATLAB code.	(b) aspect to partially unroll the loop by a factor of two.
… for j = 1:2:50 %@loop1 sum = sum + A(j); %@line1 sum = sum + A(j+1); end …	
(c) MATLAB code after weaving.	

Figure 21. Partial loop unrolling of two if the number of loop iterations is multiple of 2 using low-level advisors.

… for j = 1:1:100 %@loop1 sum = sum + A(j); end …	Intersection: - {@loop1} Advice: - transformation::unroll{factor=2}
(a) piece of MATLAB code.	(b) aspect to partially unroll the loop by a factor of two.
… for j = 1:2:100 %@loop1 sum = sum + A(j); sum = sum + A(j+1); end …	
(c) MATLAB code after weaving.	

Figure 22. Partial loop unrolling of two if the number of loop iterations is multiple of 2 using high-level advisers.

… %@loop-begin for j = 1:1:100 sum = sum + A(j); end %@loop-end …	Intersection: - {@loop-begin-@loop-end}:{for <var a1> = 1:1:<const integer c1> <body> end} :: position innermost
(a) original code.	(b) intersection based on pattern matching limited to the code region identified by the tags.

Figure 23. Example limiting the intersection of a pattern to a certain code region.

3.4 Data Types and Shapes Aspects

As MATLAB assumes matrix shapes and the type double for all the variables in the program, advanced compilers need to perform shape and type inference to produce more efficient code [11]. This is even more relevant when compiling MATLAB programs to embedded systems as they may have restrictions regarding size of storage memory, computing power, and direct

execution of operations dealing with floating-point data types. However, shape and type inference are not easy tasks. In some cases might even be impossible to determine, at compile time or without the user knowledge about the application and the scenarios related to input data, the shape and type of variables.

One advantage of our approach is that it allows users to specify concerns on the basis of shape and data types. Figure 24 shows an example where the user specifies that all variables in the program minus the variables i, j, and k, are represented by a signed fixed-point type with 32-bits of wordlength and 16 of fraction part (fixed_point(1,32,16)), and that variable j is represented as a signed integer of 16 bits.

<table>
<tr><td>...
sum = 0;
…
for j = 1:1:100
    sum = sum + A(j) * B(j+k);
end
C(i) = sum;
…</td><td>Intersection:
    - all <var v1> in ^{j,k,i}
Advice:
    - assign <v1.type> to fixed_point(1,32,16)

Intersection:
    - all <var v2> in {j}
Advice:
    - assign <v2.type> to int16</td></tr>
<tr><td>(a) piece of MATLAB code.</td><td>(b) aspects to assign data types to variables.</td></tr>
<tr><td colspan="2">...
<u>sum = fi(0, 1, 32, 16);</u> %using “fi” objects
…
for j = <u>int16(1):int16(1):int16(100)</u>
    % assuming that all data in coef and internal_state have been converted to “fi” objects
    sum = sum + A(j) * B(j+k);
end
C(i) = sum;
…</td></tr>
<tr><td colspan="2">(c) MATLAB code after weaving.</td></tr>
</table>

Figure 24. Aspects to assign data types to variables.

In some cases, one might need to use a specific set of aspects in different places where a given function is invoked, e.g., we may like to use fixed-point data types in an invocation place and floating-point data types in another invocation place. Figure 25 shows how the specification of different sets of aspects (each one in a different file) can be done.

<table>
<tr><td>...
y1=f(x1); %@f_invoke1
…
y2=f(x2); %@f_invoke2
...</td><td>Intersection:
- {@f_invoke1}
Advice:
- replace f_invoke1 with f_aspects1.asp

Intersection:
- {@f_invoke2}
Advice:
- replace f_invoke2 with f_aspects2.asp</td></tr>
<tr><td>(a) piece of MATLAB code.</td><td>(b) aspects to assign data types to variables.</td></tr>
<tr><td colspan="2">...
y1=f(x1); %@f_aspects1.asp
…
y2=f(x2); %@f_aspects1.asp
...</td></tr>
<tr><td colspan="2">(c) MATLAB code after weaving (the two files will be used by the C and MATLAB code generators).</td></tr>
</table>

Figure 25. Specifying different data-types for each function call.

The aspects related to shape and data type definitions are translated to a representation used by the back-end compilers [12]. The generators receive for each MATLAB function a specification of the shapes and data types as illustrated in Figure 26.

```
A:INT16:1x2 // a 1x2 matrix of 2 16-bit signed integer elements
B:DOUBLE:1x1 // a double element
C:INT:3x4 // a 3x4 matrix of 32-bit signed integer elements
D:FIXED_POINT(1, 10, 4):2x1 // a 2x1 matrix of fixed-point elements with 10 bits of wordlength and 4 bits of fraction
E:UINT:1x2 // a 1x2 matrix of 32-bit unsigned integer elements
```

Figure 26. Shape and data type specification used by the back-end tools.

3.5 Configuration Aspects

During the development of certain applications there might be the need to evaluate different implementations of a given function. This often leads to the emulation dynamic dispatch, which usually entails the maintenance of multiple source files or the use of pre-processing directives, such as those from the C programming language (as #ifdef, #ifndef, etc).

Figure 27 shows how an aspect rule can be used to inform the weaver to use a particular implementation called “sin1” of the function “sin”. This example has been used to explore the impact on using the following implementations of the sine trigonometric functions: the one included in MATLAB, one using a Taylor series using the first three terms, and other one using a loop-up table of 128 sin values for angles between 0 and 90 degrees. With simple substituting rules we can test the different implementations without polluting the original MATLAB code.

<table>
<tr><td>...
for z = 0:n-1
arg = 2*PI * z / n;
for k = 0:n-1
cosarg = cos(k * arg);
sinarg = sin(k * arg);
x2(z+1) = x2(z+1)+xre(k+1) * cosarg - xim(k+1)*sinarg;
y2(z+1) = y2(z+1)+xre(k+1) * sinarg + xim(k+1)*cosarg;
end
end
...</td><td>Intersection:
- <identifier i1> in {sin}
Advice:
- replace <i1.name> with “sin1”</td></tr>
<tr><td>(a) piece of MATLAB code.</td><td>(b) aspect to assign a different callee.</td></tr>
<tr><td colspan="2">...
for z = 0:n-1
arg = 2*PI * z / n;
for k = 0:n-1

cosarg = cos(k * arg);
sinarg = sin1(k * arg);
...</td></tr>
<tr><td colspan="2">(c) MATLAB code after weaving.</td></tr>
</table>

Figure 27. Aspects to specify the use of a different implementation of a function.

3.6 Design Space Exploration Aspects

In this section, we illustrate the insertion of code to explore certain aspects. In the example from Figure 28 one inserts code that will repeat the execution of a piece of code, augmenting in each execution the precision of the fixed-point variables (1 additional fraction bit per iteration). The repetitions stops when the specified number of iterations is reached or when the sum of the errors obtained in the last two iterations is below the specified value. This example is illustrated with insertion code primitives. In Figure 29 we show the same example considering higher-level aspect abstractions that directly deal with primitives for this kind of exploration.

<table>
<tr>
<td>
…
%@begin
for j = 1:1:100
sum = sum +
internal_state(j) * coef(j+ORD);
end
outa(i) = sum;
%@end
…
</td>
<td>
Intersection:
- {@begin}
Advice:
- insert { prev = zeros(1, NPOINTS);
for FRACTION = 1:1:16 % Loop to explore
quant1=quantizer('fixed', 'floor', 'wrap', [32 FRACTION]);}:: execute after

Intersection:
- {@end}
Advice:
- insert { if(abs(prev_data-outa) < 0.1)
sprintf('number of bits in fractional part: %d', FRACTION)
break;
end

prev = outa;
end
}:: execute before
</td>
</tr>
<tr>
<td>(a) piece of MATLAB code.</td>
<td>(b) aspects.</td>
</tr>
<tr>
<td colspan="2">
…
%@begin
prev = zeros(1, NPOINTS);
for FRACTION = 1:1:16 % Loop to explore
quant1=quantizer('fixed', 'floor', 'wrap', [32 FRACTION]);

for j = 1:1:100
sum = sum + internal_state(j) * coef(j+ORD);
end
outa(i) = sum;

if(abs(prev_data-outa) < 0.1)
sprintf('number of bits in fractional part: %d', FRACTION)
break;
end

prev = outa;
end
%@end
…
</td>
</tr>
<tr>
<td colspan="2">(c) MATLAB code after weaving.</td>
</tr>
</table>

Figure 28. Aspects for runtime exploration of precision with fixed-point data types using the insertion of code.

Exploration might be helpful in other domains, e.g., to determine the unrolling factor or the size of the block in the loop tilling transformation. In Figure 22 we have shown the use of aspect rules to instruct the weaver to unroll by a factor of 2 a loop. In an exploration scenario, we might consider the exploration of the unrolling factor by weaving for each unrolling factor and get feedback results (e.g., execution time, code size) in order to decide about the best unrolling factor.

<table>
<tr><td>…
%@begin
for j = 1:1:100
    sum = sum + internal_state(j) *
coef(j+ORD);
end
outa(i) = sum;
%@end
…</td><td>Intersection:
- all variables in {@begin-@end} ^{j}
Advice:
- explore variables:
fixed_point(1,32,FRACTION=1:1:16)::
execute around
Conditions:
- dynamic: {if(abs(prev_data-{outa}) <
0.1) sprintf('number of bits in fractional
part: %d', FRACTION) break; end}</td></tr>
<tr><td>(a) piece of MATLAB code.</td><td>(b) aspect.</td></tr>
<tr><td colspan="2">…
%@begin
prev = zeros(1, NPOINTS);
for FRACTION = 1:1:16 % Loop to explore
  quant1=quantizer('fixed', 'floor', 'wrap', [32 FRACTION]);

  for j = 1:1:100
    sum = sum + internal_state(j) * coef(j+ORD);
  end
  outa(i) = sum;

  if(abs(prev_data-outa) < 0.1)
    sprintf('number of bits in fractional part: %d', FRACTION)
    break;
  end

  prev = outa;
end
%@end
…</td></tr>
<tr><td colspan="2">(c) MATLAB code after weaving.</td></tr>
</table>

Figure 29. Aspects for runtime exploration of precision with fixed-point data types. In this example, all but one variables (j) are assigned to signed, 32-bit, fixed-point representation. The adviser specifies that the number of bits for the fractional part of these variables should go from 1 to 16 in steps of 1. The exploration may exit if a certain difference between the previous and the current results has been reached.

3.7 Aspects Guiding Compiler Optimizations

Our approach also focuses on aspects that specify additional information about a program that can be used to guide compilers on optimizations that are otherwise difficult or impossible to achieve, including windowing computations, data distribution and replication, distribution of computations, task level pipelining and producer/consumer patterns, streaming characteristics, etc.

4 Folding Polluting Code

The approach presented in this report will permit to fold certain concerns included by the user directly in the MATLAB program (e.g., with legacy code), which should not to be taken into

account when targeting an embedded system, for instance. These concerns may include monitoring variables, assertions, plots of data, etc. A “de-weaver" would be needed to automatically fold those aspects in aspect rules.

5 Related Work

Part of the aspect-oriented approach being researched in AMADEUS needs a code transformation engine. Several code transformation engines have been proposed. An example is the *ctt* (Code Transformation Tool) approach for transforming C code [13]. *Ctt* is a source-to-source transformation engine that uses a pattern-oriented language and allows transformation rules specified in three parts (subsections), pattern-conditions-result, which resemble the intersection-conditions-adviser used by the AOP community. It appears that *ctt* is fairly more verbose as the programmer must in essence indicate what is cut and what is paste instead.

```
PATTERN {
    description of the code selection stage
}
CONDITIONS {
    additional constraints
}
RESULT {
    description of the new code
}
```

Figure 30. The three parts for specifying a code transformation rule in ctt.

Cetus[7] [14] is an infrastructure for code transformations. It currently supports C code and uses pragmas and needs the transformations to be coded using the APIs integrated in the framework. This is a too low-level approach for users, but can be an interesting approach to add to compilers a repository of code transformations depending on the target architecture, for instance. This sort of approach is in practice only accessible to the most experience programmers or compiler writers and not the average programmer or library implementer.

Source-level code transformations are a key program transformation technique to improve specific execution metrics such as time, energy or space (memory) metrics. These transformations, however, often have conflicting goals making their choice and ordering an extremely hard optimization problem. For example, data replication increases data availability (i.e., available bandwidth) at the expense of memory storage. In addition, they are very cumbersome and error-prone for programmers to carry them manually, thus suggesting the use of automated code transformation systems and/or compilation tools. To exacerbate these difficulties, many if not all of these transformations require program knowledge that is often neither present in the original program specification nor can it be easily extracted from it. As such, compilers that use them are extremely limited by the semantics of the input programming language and thus require programmer input to guide them in the applicability of these transformations.

[7] http://cetus.ecn.purdue.edu/

Generative programming tools [15] offer a path for the automation of code transformations. They allow programmers to augment computation with key desired metric goals that will lead to the development of internal transformation application strategies. These approaches thus rely on linguistics mechanisms that are beyond the semantics of the input program languages, typically in the form of rule-based systems (e.g., pattern-apply-condition). We thus believe, to be very important to exploit the synergies between the more traditional domains of compiler optimization and code transformations holistic concepts from generative programming. Tools such as TXL [16], Tom [7], and Stratego/XL [17] may undoubtedly play an important role by enhancing the compilation flow code transformation rules and strategies.

In a technical report [18], Aslam et al describe AspectMatlab, a new language that extends MATLAB with aspect-oriented features. The design of AspectMatlab is inspired on that of AspectJ, adapted to the specific features of MATLAB. Part of the report is focused 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 in this report is that our approach keeps MATLAB sources separate from aspect-specific constructs, while AspectMatlab integrates them into a single whole. The primary advantage of AspectMatlab over our approach seems to be tighter integration between the MATLAB “base” and aspects. By contrast, our approach was designed to minimize dependencies between the MATLAB original sources and aspects. Keeping aspects separate from plain MATLAB parts provides additional guarantees of such independence. Both approaches need to deal with the future evolution of the base language. This issue 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. In the scenario of a new version of MATLAB being available providing new features, programmers can immediately reuse older sources with the new version, provided it is backwards compatible.

6 Conclusions

This report described the main aspect-oriented concepts to augment MATLAB proposed in the AMADEUS project. The report includes many examples of aspect rules considered important to monitor activity and data values, explore specific features in a MATLAB program, configure and specialize code, among others.

References

1. Gregor Kiczales, et al. *Aspect-Oriented Programming*. in *11th European Conference Object-Oriented Programming (ECOOP'97),* . 1997. Jyväskylä, Finland: Springer.
2. Kiczales, G., *Aspect-Oriented Programming.* ACM Computing Surveys (CSUR), special issue: position statements on strategic directions in computing research, 1996. **28**(4es).

3. Peri Tarr, et al., *N Degrees of Separation: Multi-Dimensional Separation of Concerns*, in *Proceedings of the 21st International Conference on Software Engineering*. 1999, ACM: Los Angeles, California, USA. p. 107-119.
4. Robert E. Filman and D.P. Friedman, *Aspect-Oriented Programming is Quantification and Obliviousness*, in *Workshop on Advanced Separation of Concerns, at OOPSLA'00*. 2000: Minneapolis, USA.
5. D. L. Parnas, *On the Criteria to be Used in Decomposing Systems into Modules.* Communications of the ACM 1972. **15**(12): p. 1053-1058.
6. João M. P. Cardoso, João M. Fernandes, and M. Monteiro, *Adding Aspect-Oriented Features to MATLAB*, in *SPLAT! 2006, Software Engineering Properties of Languages and Aspect Technologies, a workshop affiliated with AOSD'2006*. 2006: Bonn, Germany.
7. Emilie Balland, et al. *Tom: Piggybacking rewriting on java*. in *18th Conference on Rewriting Techniques and Applications (RTA'07)*. 2007: Springer Berlin / Heidelberg.
8. Emilie Balland, Pierre-Etienne Moreau, and A. Reilles, *Rewriting Strategies in Java.* Electronic Notes in Theoretical Computer Science (ENTCS) 2008. **219**: p. 97-111.
9. Joseph D. Gradecki and N. Lesiecki, *Mastering AspectJ: Aspect-Oriented Programming in Java*. 2003, New York, NY, USA John Wiley & Sons, Inc. 432.
10. Ricardo Nobre and P. Maia, *MATLAB to TOM-IR Front-End*. 2009, Instituto Superior Técnico (IST), Universidade Técnica de Lisboa (UTL).
11. Pramod G. Joisha and Prithviraj Banerjee, *An algebraic array shape inference system for MATLAB©.* ACM Transactions on Programming Languages and Systems (TOPLAS), 2006. **28**(5): p. 848-907.
12. Ricardo Nobre and P. Maia, *MATLAB to C Generator*. 2009, Instituto Superior Técnico (IST), Universidade Técnica de Lisboa (UTL).
13. Maarten Boekhold, et al., *A Programmable ANSI C Transformation Engine*, in *Proceedings of the 8th International Conference on Compiler Construction, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99*. 1999, Springer Berlin / Heidelberg. p. 292-295
14. Sang-Ik Lee, Troy A. Johnson, and R. Eigenmann, *Cetus An Extensible Compiler Infrastructure for Source-to-Source Transformation* in *Proceedings of the 16th Annual Workshop on Languages and Compilers for Parallel Computing (LCPC'2003)*. 2003. p. 539-553.
15. Krysztof Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. June 16, 2000: Addison-Wesley Professional.
16. Cordy, J.R., *The TXL Source Transformation Language.* Science of Computer Programming 2006. **61**(3): p. 190-210.
17. Martin Bravenboer, et al., *Stratego/XT 0.17. A language and toolset for program transformation.* Science of Computer Programming, Elsevier 2008. **72**(1-2): p. 52-70.
18. Aslam T., Doherty J., Dubrau A., Hendren L. AspectMatlab: An Aspect-Oriented Scientific Programming Language. Sable Technical Report No. sable-2009-03.

Appendix A: Glossary

Advice: name used in the context of many AOP languages to refer to the actions to be performed when a joinpoint is reached during the execution of a program. In AspectJ, advice are nameless blocks of code but in other AOP languages advice can be plain methods. Instead of being called explicity, as with methods in traditional OOP, advice are in general called *implicitly*, upon reaching one of the specified joinpoints.

AMADEUS project (ASPECTS AND COMPILER OPTIMIZATIONS FOR MATLAB SYSTEM DEVELOPMENT**)**: A research project partially funded by the Portuguese science foundation (FCT): POCTI, PTDC/EIA/70271/2006. AMADEUS involves research groups from FEUP (Faculdade de Engenharia da Universidade do Porto), INESC-ID/IST (Lisbon), Universidade do Minho (Braga), and the Universidade Nova de Lisboa (UNL). The AMADEUS project focuses on aspect-oriented techniques to enhance MATLAB programming and to assist MATLAB to C transformations.

Annotation: text usually embedded in code for a specific purpose. Usually annotations can be represented as pragmas in programming languages with pragma support or embedded as comments using certain marks (e.g., '@') to distinguish those annotations from usual comments.

Aspect (or aspect module): a crosscutting concern modularized. An aspect is a special kind of module enclosing functionality that otherwise would cut across multiple modules of a system. Aspects have novel kinds of features that endow them with the compositional capability to compose its crosscutting functionality without compromising code locality and other modularity properties.

Aspect rules: in the context of AMADEUS, aspect rules are used to identify the rules that enable the use of an aspect.

Attributes: in the context of AMADEUS, attributes are used to identify the set of values that define properties of an aspect after executed (e.g., if it has intersected and how many times, if the action has been applied).

Back-end: term broadly used in compilers to identify the final stages of the compilation which usually involves target-specific optimizations and code generation.

Code instrumentation: the insertion of code mechanisms to monitor certain aspects of the program.

Code transformation strategy: a specification of the set of transformations and the ordering to apply them to the program code.

Code specialization: modification in the program code in order to reflect target-specific properties.

Concern: any kind of facet, functionality or property in a software system that developers may want to consider as a separate concept and want to see represented separately from the other concepts – in its own module – to better reason with it and and manipulate it.

Conditions: Boolean expressions used to specify when an advice/action should be trigger.

Crosscutting concern: a concern that does not align with the (module) decomposition of the system at hand. Non aligning concerns tend to cut across the modules of the system and often give rise to the tangling and scattering symptoms.

Data shape: used to refer the dimensions of the matrix variables used in a MATLAB program.

Design space exploration (DSE): the exploration of optimization options (or possible characteristics of a given solution) in order to search for better implementations.

Execution analysis: analysis of what happens during execution of a program.

Front-end: term broadly used in compilers to identify the first stage of the compilation which involve lexical, syntactic, semantic analysis, and the generation of intermediate representations (IR) of the input program.

Intermediate representation (IR): a data structure to represent an input program that is used to perform stages in the compilation flow such as optimizations and code generation. Depending on the goals, the IR can be represented as a tree, as a graph, or a list of instructions.

Intersection: concept akin to that of pointcut used in the context of AMADEUS.

Joinpoint: originally defined as *a principled point in the execution of a program*. A joinpoint is a well-defined event in the execution of a program, such as the call to a method, the access to an object field, the execution of constructor, or the throwing of an exception. The execution trace of a program can be approached as a sequence of such events. Some joinpoints are atomic in that no other joinpoint can originate between the beginning of the joinpoint and its conclusion (e.g., “field get” and “field set”). Other joinpoints have nested joinpoints (e.g., “method execution” joinpoints). Joinpoints are always properly nested: two joinpoints are either disjoint or one is included in the other.

Joinpoint model: one of dimensions in the design of an aspect-oriented language. The joinpoint model of an AOP language characterizes that language and determines the kinds of quantifications allowed in that language. The level of obliviousness attained directly depends of the quantification capabilities of the language.

Joinpoint shadow: points, regions or locations in the static representation of the program (usually source code) that represent joinpoints. The joinpoint shadows are the actual points where the compiler actually operates.

Obliviousness: the possibility of an aspect module to add, change or delete functionality and/or behavior of another module, or set of modules, in a way that code from the other modules does not depend on the first module. The other modules are said to be oblivious to

the aspect module. Compiling the other modules without the aspect module should not generate compiler errors. **Programmer obliviousness** is the possibility of an aspect module to add, change or delete functionality and/or behavior of a set of modules that were developed by programmers that were oblivious to the existence of the aspect module.

Pattern matching engine: a computing engine to determine the similarity between two or more patterns. Patterns can be represented as graphs, trees, strings, regular expressions, etc.

Pointcut: a declarative clause that specifies sets of joinpoints. As the places in the source code relating to the specified joinpoints (i.e., joinpoint shadows) are non-contiguous, the set of *captured joinpoints* cuts across the system's module structure. Many AOP languages provide pointcuts with the ability to capture useful values from the context of the joinpoint, such as method arguments, the reference to the currently executing object, or the target of a method call. In some AOP languages, certain pointcuts serve to restrict the set of joinpoints captured by other pointcuts, giving rise to a form of constraint programming. Technically, logic programming can also be used to yield highly expressive and powerful forms of pointcut protocols.

Program-augmentation: the addition of complementary information to a given program.

Quantification: the expressing of a condition or assertion over a set of program properties and the ability to perform specific actions on the elements derived from those assertions. In the general case, quantification is the ability to state: "*In programs* P*, whenever condition* C *arises, perform action* A." Different kinds of quantification map to different joinpoint models.

Rewriting (code rewriting/ program rewriting): changes in the original code in order to accomplish certain goals.

Rewriting strategy: see code transformation strategy.

Scattering: a symptom in program source code frequently observable when in the presence of one (or several) crosscutting concern(s). A concern is *scattered* when its associated source code of a module is not modularized but instead spread over multiple modules.

Tangling (in source code): a symptom in program source code frequently observable when in the presence of one (or several) crosscutting concern(s). Tangled code is code whose parts relate to more than one concern but are nevertheless enclosed within the same unit of modularity, harming comprehensibility and other software engineering properties.

Unit of modularity: any element of a program that has an interface and that can be handled through its interface. Units of modularity include all kinds of modules (such as classes in OOP) and some units that bear some of its characteristics but are not modules (such as class methods in OOP and procedures in procedural programming).

Weaving: the phase during which aspect functionality is composed with the remaining modules of the system. The exact moment when composition takes place (e.g., static time, load time, run time) and how that impacts on language mechanisms depend on the language/tool design and the implementation technologies.

Appendix B: Intersections

Rules

Below are a first set of grammar rules for the intersections:

1. Start → [<NAME OF FILE>”.”][<NAME OF FUNCTION>”.”] JoinPoint “:” [AdvancedDescriptor] Descriptor
2. JoinPoint → “{“ Tags “}”
3. Tags → CodeSegment | Tag (“,”CodeSegment | Tag)*
4. CodeSegment → <@Tag> “-“ <@Tag>
5. Tag → <@Tag>
6. AdvancedDescriptor → PATTERN LANGUAGE (grammar to be defined)
7. Descriptor → (“all” | “all uses” | “all definitions”) Variables [“in“ IdentifierSet]
8. IdentifierSet → “{“ <IDENTIFIER> (“,” <IDENTIFIER>)* “}” |
   “{“ “*” “}”
9. Variables → “<var” [<IDENTIFIER>] ”>” |
   “<ident” [<IDENTIFIER>] “>” |
   “<key” [<IDENTIFIER>] ”>” |
   “<tag” [<IDENTIFIER>] ”>” |
   “<const” [“integer” | “real”] [<IDENTIFIER>] “>”

Examples of intersections in a given code

- A tag referring a line in the program:

{@here1} // line in the code where this tag is located

{@here1} : <ident> in {sum} // identifier “sum” in the line in the code where this tag is located

{@here1}: <ident> in {*} // all identifiers in the line in the code where this tag is located

{@here1}: <var> in {sum} // variable “sum” in the line in the code where this tag is located

{@here1}: <var> in {*} // all variable in the line in the code where this tag is located

{@here1}: <key> in {for} // keyword “for” in the line in the code where this tag is located

{@here1}: <key> in {*} // all keywords in the line in the code where this tag is located

- Two tags referring segments of code in the program

{@begin-@end} // segment of code

{@begin-@end}: all uses <var> in {a, sum} // all uses of variables “a” and “sum” in the segment of code

{@begin-@end}: all uses <var> in {*} // all uses of variables in the segment of code

{@begin-@end}: all definitions <var> in {*} // all definitions of variables in the segment of code

{@begin-@end}: all uses <ident> in {a, sum} // all uses of identifiers “a” and “sum” in the segment of code

{@begin-@end}: all uses <ident> in {*} // all uses of identifiers in the segment of code

{@begin-@end}: all <key> in {for, while} // all uses of keywords “for” and “while” in the segment of code

{@begin-@end}: all uses <key> in {*} // all keywords used in the segment of code

- Identifying the file and the function where intersections may occur:

myFile.myFunction.{@begin-@end} // segment of code in function “myFunction” located in file “myFile”

myFunction.{@begin-@end} // segment of code in function “myFunction”

- Examples referring tags:

{@begin-@end}: <tag> in {@here} // line in the segment of code where tag @here is located

{@begin-@end}: <tag> in {*} // lines in the segment of code where tags @ are located

<tag> in {*} // all lines in the code where tags @ are located

<tag> in {@here1} // line in the code where tag @here1 is located (this is equivalent to {@here1} but adds also a mechanism to the adviser to use the tag labels in actions by using <tag> in a template, for instance)

{@begin-@end}: all <key> in {for, while} // all uses of keywords “for” and “while” in the segment of code

{@begin-@end}: <key> in {*} // all keywords used in the segment of code

- Identifying the file and the function where intersections may occur:

myFile.myFunction.{@begin-@end} // segment of code in function “myFunction” located in file “myFile”

myFunction.{@begin-@end} // segment of code in function “myFunction”

Rules for Advices/Actions

(not complete)

replace “<” <IDENTIFIER>[.name | .type | .value] “>” with <STRING>

execute before | after | around | (before , after) | (after , before)