Adding Aspect-Oriented Features to MATLAB

João M. P. Cardoso
Universidade do Algarve
Campus de Gambelas
8000-117, Faro
INESC-ID, 1000-029, Lisboa
PORTUGAL
jmpc@acm.org

João M. Fernandes
Dep. Informática / CCTC
Universidade do Minho
Campus de Gualtar
4710-057 Braga
PORTUGAL
jmf@di.uminho.pt

Miguel P. Monteiro
Escola Superior de Tecnologia
Instit. Politécnico de Castelo Branco
Avenida do Empresário
6000-767 Castelo Branco
PORTUGAL
mmonteiro@uminho.pt

ABSTRACT

This paper presents an approach to enrich MATLAB with aspect-oriented extensions to experiment different implementation features. The language we propose aims to configure the low-level data representation of real variables and expressions, to a specifically-tailored fixed-point data representation that benefits from a more efficient support by computing engines (e.g., DSPs, application-specific architectures, etc.) without specific hardware-based floating point units. Additionally, the approach aims to help developers to introduce handlers and monitoring features, and to configure a function with an optimized implementation.

Keywords

Aspect-Oriented Programming, MATLAB.

1. INTRODUCTION

MATLAB is an interpreted, imperative programming language mainly based on matrix data types and operations on them. It is widely used in scientific computing, control systems, signal processing, image processing, system engineering, simulation, etc. Mathworks[1], the company proprietary of the language, furnishes a complete integrated environment to develop MATLAB projects. The environment includes a number of suitable debug features. Also included is Simulink, a visual, component based, environment also using MATLAB, suitable for simulation of discrete and continuous systems. Several toolboxes (packages) exist that include special functions and features in a number of domains. Such packages make the language one of the preferred choices to model and simulate complex systems. More than 800 books[2] dedicated to MATLAB attest to its wide adoption.

Like most interpreted languages (e.g., Perl, Python, etc.), MATLAB does not require the declaration of variables. By default, the numeric representation used is the floating-point data type with double precision (64 bits, according to the IEEE standard 754 format). Other types of numeric data are integer (with 8, 16, 32 and 64 bits) and single precision floating-point data types[3]. MATLAB supports other numeric representations by using specific toolboxes. They enable the assignment of certain data-types and operation properties (e.g. overflow mode) to MATLAB variables. Useful features of MATLAB include operator overloading, function polymorphism and dynamic type specialization. Function polymorphism enables the same function to be called with different numbers and types of arguments. Dynamic type specialization enables variables to represent different data types during runtime. For instance, we can simulate the same code by applying stimulus with different data types.

The above features can really help to explore certain behaviors and data types by simulation. However, they are extremely cumbersome, error-prone and tedious for tasks such as exploiting non-uniform fixed-point representations, monitoring certain variables during a timing window, or to include handlers to watch specific behaviors. Each time these kinds of features are necessary, one needs to perform invasive changes on the original code, as well as to insert new code. This problem is felt in other implementation issues as well, since MATLAB can be regarded as a specification rather than an implementation language. There are open issues related to automatic synthesis of MATLAB specifications to a software language or a hardware description language [1]. Those issues heavily rely on attaining a given desired efficiency level. Various research efforts attempting to automate certain implementation issues took place in the past. For instance, the transformation from floating- to fixed-point data formats was conducted with some restrictions to MATLAB specifications [2]. However, it is usually claimed that the developer should have full control of the process since automatic conversions are not trivial and must be efficient. Even in this case, some automatic support is still necessary to assist developers during implementation phases.

In this paper, we propose aspect-oriented extensions to the MATLAB language in order to help system modeling and exploration of certain features conceiving system implementation. Our approach heavily relies on the separation of concerns [3] (data types versus behaviors). One of the advantages is related to the fact that 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. Furthermore, the extensions we propose can be used in other languages as well.

The rest of the paper is organized as follows. Section 2 describes the main motivation for our work. Section 3 presents the approach, by providing some examples of aspects. Section 4 compares our approach to related work. Finally, concluding remarks are presented in section 5.

[1] The Mathworks Inc., http://www.mathworks.com
[2] http://www.mathworks.com/support/books/index.html
[3] http://www.mathworks.com/access/helpdesk/help/techdoc/ matlab.html

2. MOTIVATION

Certain implementation requirements entail the use of sufficient bit-widths to represent numeric data (integer and real numbers), achieving an acceptable accuracy. Bit-widths can be exploited, e.g. to save resources and speedup performance through specialized and lower latency arithmetic operators [4] or through sub-word level parallelism [5].

Fixed-point arithmetic is usually the preferred implementation of several digital signal-processing systems due to their efficient support when targeting *Digital Signal Processors* (DSPs) without hardware floating-point units and specific hardware, e.g., *Field-Programmable Gate Arrays* (FPGA) implementations. Specific architectures may also use specialized data-types (e.g. floating point arithmetic over data types not defined by the IEEE 754 standard). Those implementations need several tests to acquire the necessary bit-widths in order to have the required accuracy (acceptable quantization errors) and required behavior. Several authors proposed methods to automatically translate floating- to fixed-point representations [6][7][8][9][10]. Some methods rely on profiling, while other methods rely on static schemes. Although this is a very important topic, the translation usually serves to help the designer only, since neither those methods are fully automatic nor can be applied without restrictions. In certain cases, designer experience and knowledge of the system requirements (which may include more than accuracy requirements, for instance related to dynamic range or precision) is the determining factor to the success of the final implementation. Therefore, simulation and specification refinement still play an important role at both data and behavioral levels.

As mentioned above, the MATLAB environment includes special packages to deal with fixed-point representations. MATLAB supports two toolboxes for fixed-point computations: Filter Design Toolbox and Fixed-Point Toolbox. The Filter Design toolbox furnishes functions to quantize values represented as, e.g., doubles in fixed-point representations (*quantizer* and *quantize*). The Fixed-Point Toolbox furnishes fixed-point data types and functions. *Fi* objects can be defined to represent a number of fixed-point properties and can be associated to variables and to arithmetic operations.

Certain exploration features need changes in the code to be accomplished. Such changes are error-prone, tedious, and difficult to maintain. Sometimes, changes require manual refactoring of large sections of code. Examples of changes are insertion of code statements, new function arguments, different data-types, and global variable definitions. Often, the developer must manage multiple versions of the specification, which usually gives rise to additional problems when changes must be made.

Consider as an example the MATLAB code illustrated in Figure 1. It represents an algorithm to perform the *Discrete Fourier Transform* (DFT) – widely used in signal processing systems. The function can be tested using the test program shown in Figure 2. To test it with uniform fixed-point data types, we merely need to add a line of MATLAB to the test program (see Figure 3). However, to test the function using fixed-point representations with specialization of each variable and operation, we need to change the original function. Figure 4 represents the changed function. Note that the fixed-point representations used in the example are included here as a general example and have not been necessarily exploited to fulfill a specific accuracy or behavior.

During design phases we usually need models that closely resemble implementation details. As an example, if a specific hardware implementation is needed, results with fixed-point numeric representations might be necessary to validate the final implementation using a comparison between *Hardware Description Language* (HDL) and MATLAB simulations. Modeling with specialized fixed-point representations is of great importance since such implementations are usually needed to satisfy various requirements, namely low power dissipation, low energy consumption, better performance and fewer hardware resources.

Notice that this kind of data type specialization is also needed when object-oriented programming is used. In that case, even if you use specific built-in class support for fixed-point data types, there is always the need to directly specify the data type specializations required. Even though we do not focus the object-oriented case, we believe our approach can also be used in that context.

```
function [y] = dft(x)
y=zeros(size(x));
N=length(x);
t=(0:N-1)/N;
for k=1:N
   y(k) = sum(x.*exp(-j*2*pi*(k-1)*t));
End
```

Figure 1: Simple MATLAB example (function to perform a Discrete Fourier Transform, source: [1]) – original code.

```
function [y] = dft_specialized(x)
y=zeros(size(x));
N=length(x);
t=(0:N-1)/N;
quant1=quantizer('fixed','floor','wrap', [18 16]);
t=quantize(quant1, t);
quant2=quantizer('fixed','floor','wrap', [23 20]);
pi_fix = quantize(quant2, pi);
quant3=quantizer('fixed', 'floor', 'wrap', [20 8]);
quant4=quantizer('fixed','floor', 'wrap', [23 10]);
quant5=quantizer('fixed','floor', 'wrap', [24 10]);
quant6=quantizer('fixed','floor', 'wrap', [26 12]);
quant7=quantizer('fixed','floor', 'wrap', [28 14]);
quant8=quantizer('fixed','floor', 'wrap', [32 16]);
for k=1:N
   v1 = quantize(quant3, (k-1)*t);
   v2 = quantize(quant4, pi_fix*v1);
   v3 = quantize(quant5, -j*2*v2);
   v4 = quantize(quant6, exp(v3));
   v5 = quantize(quant7, x.*v4);
   y(k) = quantize(quant8, sum(v5));
end
```

Figure 2: Simple MATLAB example – code needed to model specialized fixed-point bit-widths.

```
function testdft;
x=[1 2 3 4 5 6 7 8];

dft(x)
```

Figure 3: Example of MATLAB – Test of dft function with double-precision data types.

Sometimes, there is also the need to keep different implementations of a given function. As an example, consider arithmetic division that can be implemented with look-up tables, iterative algorithms, a combinatorial divisor, etc. Each implementation may affect the overall accuracy of the system and therefore requires modeling prior to implementation. This entails changes in the original code and ultimately maintenance of multiple versions of the code. Configuration features ameliorate the problem, since with a rule one can specify the implementation used by a simulation in a given development phase.

```
function testdft;
x=[1 2 3 4 5 6 7 8];

x=fi(x, 1, 9, 5); %new line

dft(x)
```

Figure 4: Example of MATLAB – Test of dft function with a uniform fixed-point representation.

3. ASPECT-ORIENTED EXTENSIONS

Our approach envisages the usage of two separate parts (source files) to model a given system: MATLAB code representing the primary behavior and aspect-oriented rules. Aspect-oriented rules are mainly used to reassign data types to variables in the MATLAB code, to introduce handlers and monitoring features, and to configure a function with a given implementation. The rules aim to facilitate development of systems that require refinement of specific features needed for implementation of the original specification. The proposed rules have declarative semantic as opposed to the imperative semantic of MATLAB. According to their semantics, rules can be divided in the following groups:

- **Monitor rules** help users to observe the runtime characteristics of MATLAB variables. They include special behavior related to monitoring, such as return the maximum value of a certain variable during the simulation period.
- **Handler rules** are a kind of assertions with the purpose to ensure that certain conditions hold during the simulation period.
- **Type assignment rules** are used to bind different types to the variables of the MATLAB specification.
- **Configuration rules** are used to statically bind a different implementation to a given function or operator.

Figure 5 presents the outline of the proposed system. We propose that aspect-oriented rules and MATLAB code be specified in different files. A transformation engine (weaver) is responsible to generate new MATLAB code that includes the features composed by the specified aspect-oriented rules.

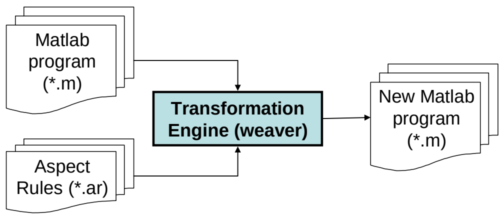

Figure 5: Outline of the MATLAB-based system enhanced with aspect-oriented rules.

Type assignment rules are one of the most important aspects of our approach. Using MATLAB, users start with a specification using double precision floating-point data-types (the default MATLAB numeric data type). The following example shows a simple MATLAB code to multiply two variables, previously assigned to constants:

```
a = 2;
b = 3;
c = a*b;
```

All the assigned and calculated values of the above example are represented as doubles. If we need to test the code with integer data types, e.g. of 16 bits, the original code must be changed to:

```
a = int16(2);
b = int16(3);
c = int16(a*b);
```

Using our approach the original code is maintained and we only need to add an assignment rule in the aspect part:

```
For all variable in program do: set type int16;
```

This would tell the transformation engine to add to the original code the code needed to assign the type *int16* for each variable. Moreover, if we need to simulate the original code using different data types for each variable, we only need to use the following rules:

```
For variable a in program do: set type int16;
For variable b in program do: set type int16;
For variable c in program do: set type int32;
```

In this case we are specifying the following MATLAB code:

```
a = int16(2);
b = int16(3);
c = int32(a*b);
```

This kind of aspect-oriented rules may require the decomposition of arithmetic expressions onto sub-expressions in order to apply different rules to each sub-expression. Suppose the existence of the following statement in a MATLAB specification:

```
a = b*c+d;
```

To bind different specialized fixed-point representations to the sub-expressions that will be computed by this statement, we would need to change the original code to:

```
v1 = b*c;
a = v1+d;
```

Then, each variable in the above assignments can be bound to a specific fixed-point data representation. Although this is a straightforward step, it requires changes in the original code, making it less legible. To address this problem, we include in the aspect features a decomposition rule telling the transformation engine to decompose a given expression into the specified sub-expressions. An example of this kind of rule is:

```
for "a = b*c + d;" :
   decompose { v1=b*c; v2=v1+d; a=v2; };
```

This way, we may now include type assignment rules to each variable (a, b, c, d, v1 and v2). Note that the statements between brackets in the decompose command should be pure-MATLAB with the same behavior as the original expression.

Monitor type rules may help developers by including observing behavior without changing the original MATLAB code. Examples of monitors are to outputting to a file the values of a specific variable during simulation (e.g., `For variable a in program do: print("a.dat", a);`). Sometimes it is also required to observe the maximum and minimum values assigned to a given variable in the program. This is usually required when exploring bit-width analysis since it may expose the number of bits to represent a certain variable. Adding this behavior to the original code may require the use of global variables and the addition of specific code to compute the maximum and minimum values for each assignment. Note also that usually this behavior is auxiliary and is latter removed. With our aspect-oriented rules, such behavior is kept separate from the original MATLAB code (e.g., `For variable A in program do: print(ecran, a:max);`).

Handler type rules can also help developers observing the occurrence of certain values in variables of the program (e.g., `if module1: a > 100 {print(ecran, "warning: value of module1:a exceeds: "+100);}`). Note that handler rules are similar to assertions.

Finally, configuration rules are used to assign to an operator (arithmetic or logical) or a function an implementation different from the original one (e.g., `Use for "module1":"f1": configuration "my_f1";`).

Based on the initial example illustrated in Figure 1, we show in Figure 6 an example of rules to bind all variables of the original "dft" function to a fixed-point uniform representation of <1, 10, 5> (10-bit signed fixed-point representation, using 5 bits in the fractional part). Figure 7 shows an example of rules to bind each operand of the "dft" function with a specialized fixed-point representation according to the result shown in Figure 2. Note that expressions already decomposed in the original code do not need decomposition commands in the aspect-oriented rules.

```
Rule 1 : type is assignment {
   Typedef fixed1 = fixed<1, 10, 5>;

   For all variable in dft do:
      set type fixed1;
}
```

Figure 6: Quantification rules applied to the function presented in Figure 1 for uniform fixed-point representation.

```
Rule 1 : type is assignment {
   For "y(k)=sum(x.*exp(-j*2*pi*(k-1)*t));":
      decompose {
         v1=(k-1)*t;
         v2=pi_fix*v1;
         v3=-j*2*v2;
         v4=exp(v3);
         v5= x.*v4;
         y(k)=sum(v5);
      };
   set fixed = {overflow="wrap"; round="floor"};
   Typedef fixed1 = fixed<1, 18, 16>;
   Typedef fixed2 = fixed<1, 23, 20>;
   Typedef fixed3 = fixed<1, 20, 8>;
   Typedef fixed4 = fixed<1, 23, 10>;
   Typedef fixed5 = fixed<1, 24, 10>;
   Typedef fixed6 = fixed<1, 26, 12>;
   Typedef fixed7 = fixed<1, 28, 14>;
   Typedef fixed8 = fixed<1, 32, 16>;
   For variable t in dft do: set type fixed1;
   For variable pi in dft do: set type fixed2;
   For variable v1 in dft do: set type fixed3;
   For variable v2 in dft do: set type fixed4;
   For variable v3 in dft do: set type fixed5;
   For variable v4 in dft do: set type fixed6;
   For variable v5 in dft do: set type fixed7;
   For variable v6 in dft do: set type fixed8;
   For variable v7 in dft do: set type fixed9;
   For variable v8 in dft do: set type fixed10;
}
```

Figure 7: Quantification rules applied to the function presented in Figure 1 for variable (specialized) fixed-point representation.

Each rule may have one or more commands. The set of commands for each rule is considered in the sequential order in which they appear in the aspect-oriented rule's files. In the case of overlapping conflicts in commands, the last command prevails. Figure 8 shows some examples of the proposed rules.

```
Apply Rule1; // several rules may be applied:
             // Apply rule1:rule2:rule3;

Rule 1 : type is Monitor {
   Set myVars1 = {a, b, c};
   For each variable A in program do:
      print(ecran, A: value for each change:);
   For each variable A in myVars1 do:
      print(file:"data.txt", A);
   For variable A in program do:
      print(ecran, A:max); // mean, abs, etc.
   For each variable A in myVars1 do:
      print(ecran, A:min);
   For each variable A in module1 do:
      print(ecran, max:A);
}

Rule 2 : type is assignment {
   Typedef fixed1 = fixed<1, 10, 4>;
   set float={precision="single"};
   For all variable in program do:
      set type float;
   For all variable in "module2" do:
      set type fixed1;
   For all variable in "module3" do:
      set type fixed1;
}

Rule 3 : type is handler {
   If "module1":A > 100 {
      print(ecran,
         "warning: value of A exceeds: "+100);
   }
}

Rule 4 : type is configuration {
   use for "function1": configuration "func1";
   use for "module1":"/": configuration "myDIV";
}
```

Figure 8: Examples of aspect-oriented rules (words in italic represent user definitions).

4. RELATED WORK

To the best of our knowledge, this is the first approach to consider aspect-oriented rules to assign numeric data types to a MATLAB specification.

In [2], 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 [11]. In AML, the primary behavior is written with a MATLAB-like language. AML allows the programmer to write annotations that represent properties of sparse matrices, in completely separated way 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 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. AML was first described as an aspect-oriented system but is no longer considered as such [12].

Our proposal differs from the one above in that although type refinement may help compilers to produce optimized code, the aspects we present are intended to help developers to model and to explore different possible implementations of a given MATLAB specification without changing the original code and without the need to manage multiple MATLAB specifications. Moreover, most aspects we propose would be unsuitable to embed in the original specification as a form of annotations. There are various reasons for that. First, that would be responsible to a less legible code and would impose difficulties whenever changes in the original code are required. Second, that would still require more than one version of the MATLAB specification when we need to explore different data types for a given variable. Third, some of the rules are intended to be applied globally, not just to a certain function. With our approach explorations can be performed with the same MATLAB specifications by simply employing different aspect-oriented rules. Our approach uses a declarative type of aspect semantics suitable to be applied both local and globally.

5. CONCLUSION

This paper presents an approach to add aspect-oriented rules to MATLAB specifications in order to help developers to explore implementation features related to numeric data type representations. The approach uses a separation of concerns concept such that MATLAB behavior and aspect-oriented rules (e.g., numeric data type assignments) are separately specified. We believe the approach brings significant benefits as it enables developers to explore numeric data type representations without changing the original MATLAB code.

Work in progress includes studies about other aspect-oriented rules and trying the approach with other programming languages. Further plans include the specification of a grammar for the aspect-oriented rules and experiments on the implementation of the transformation engine.

ACKNOWLEDGEMENTS

João M. Fernandes and Miguel P. Monteiro are partially supported by FCT under projects PPC-VM (POSI/CHS/47158/2002) and SOFTAS (POSI/EIA/ 60189/2004).

REFERENCES

[1] R. Allen, Compiling High-Level Languages to DSPs, in *IEEE Signal Processing Magazine*, vol. 22, no. 3, pp. 47-56, May 2005.

[2] J. Irwin, J.-M. Loingtier, J. Gilbert, G. Kiczales, J. Lamping, A. Mendhekar and T. Shpeisman, Aspect-Oriented Programming of Sparse Matrix Code, in *Int'l Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'97)*, Marina del Rey, California, USA, LNCS 1343, Springer, pp. 249-256, December 1997.

[3] D. L. Parnas, On the criteria to be used in decomposing systems into modules, in *Communications of the ACM*, vol. 15, no. 12, pp. 1053-1059, December 1972.

[4] S. Mahlke, R. Ravindran, M. Schlansker, R. Schreiber, and T. Sherwood, Bitwidth Cognizant Architecture Synthesis of Custom Hardware Accelerators, in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Synthesis*, vol. 20, no. 11, pp. 1355-1371, November 2001.

[5] K. Scott, and J. Davidson, Exploring the Limits of Sub-Word Level Parallelism, in *9th Int'l Conference on Parallel Architectures and Compilation Techniques (PACT'00)*, pp. 81-91, October 2000.

[6] M. L. Chang, S. Hauck, Précis: A Usercentric Word-Length Optimization Tool, in *IEEE Design and Test of Computers*, vol. 22, no. 4, pp. 349-361, July/August 2005.

[7] D-U. Lee, A. A. Gaffar, R. C. C. Cheung, O. Mencer, W. Luk, and G. A. Constantinides, Accuracy guaranteed bit-width optimization, to appear in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.*

[8] A. Nayak, M. Haldar, A. Choudhary, and P. Banerjee, Precision and error analysis of MATLAB applications during automated hardware synthesis for FPGAs, in *DATE 2001*, pp. 722-728, March 2001.

[9] S. Roy, and P. Banerjee, An Algorithm for Trading Off Quantization Error with Hardware Resources for MATLAB-Based FPGA Design, in *IEEE Transactions on Computers*, vol. 54, pp. 886-896, July 2005.

[10] P. Banerjee, D. Bagchi, M. Haldar, A. Nayak, V. Kim, and R. Uribe, Automatic Conversion of Floating-point MATLAB Programs into Fixed-point FPGA based Hardware Design, In *41st Annual Conference on Design Automation (DAC'04)*, pp. 484-487, June 2004.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, Lopes, C. V., Loingtier, J.-M., Irwin, J. Aspect Oriented Programming, *ECOOP'97*, Jyväskylä, Finnland, June 1997.

[12] C. V. Lopes, AOP: A Historical Perspective (What's in a Name?). In Aspect-Oriented Software Development, pp. 97–122. Addison-Wesley, 2005.