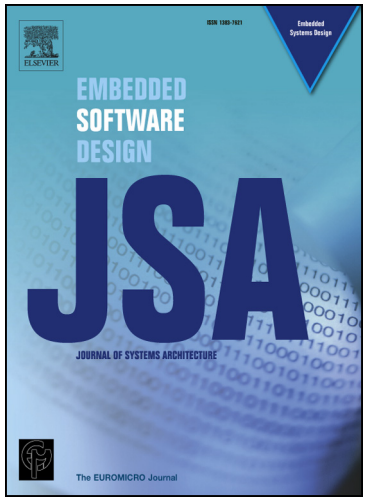

Enriching MATLAB with aspect-oriented features for developing embedded systems

João M.P. Cardoso, João M. Fernandes, Miguel P. Monteiro, Tiago Carvalho, Ricardo Nobre

PII: S1383-7621(13)00052-0
DOI: http://dx.doi.org/10.1016/j.sysarc.2013.04.003
Reference: SYSARC 1108

To appear in: *Journal of Systems Architecture*

Received Date: 7 November 2012
Revised Date: 28 February 2013
Accepted Date: 16 April 2013

Please cite this article as: J.M.P. Cardoso, J.M. Fernandes, M.P. Monteiro, T. Carvalho, R. Nobre, Enriching MATLAB with aspect-oriented features for developing embedded systems, *Journal of Systems Architecture* (2013), doi: http://dx.doi.org/10.1016/j.sysarc.2013.04.003

Enriching MATLAB with aspect-oriented features for developing embedded systems

João M. P. Cardoso[1], João M. Fernandes[2], Miguel P. Monteiro[3],
Tiago Carvalho[1], and Ricardo Nobre[1]

[1]Universidade do Porto
Faculdade de Engenharia/Dep. Eng. Informática
Rua Dr. Roberto Frias, 4200-465 Porto, Portugal
jmpc@acm.org, tiago.carvalho@fe.up.pt, ricardo.nobre@fe.up.pt

[2]Universidade do Minho
Dep. Informática / Centro Algoritmi
Campus de Gualtar, 4710-435 Braga, Portugal
jmf@di.uminho.pt

[3]Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia, Dep. Informática,
Quinta da Torre, 2829-516 Caparica, Portugal
mtpm@fct.unl.pt

ABSTRACT

This article presents an approach to enrich the MATLAB[1] language with aspect-oriented modularity features, enabling developers to experiment different implementation characteristics and to acquire runtime data and traces without polluting their base MATLAB code. We propose a language through which programmers configure the low-level data representation of variables and expressions. Examples include specifically-tailored fixed-point data representations leading to more efficient support for the underlying hardware, e.g., digital signal processors and application-specific architectures, without built-in floating point units. This approach assists developers in adding handlers and monitoring features in a non-invasive way as well as configuring MATLAB functions with optimized implementations. Different aspect modules can be used to retarget common MATLAB code bases for different purposes and implementations. We validate the proposed approach with a set of representative examples where we attain a simple way to explore a number of properties. Experiment results and collected aspect-oriented software metrics lend support to the claims on its usefulness.

Keywords

Aspect-Oriented Programming, MATLAB, Embedded Systems.

1. INTRODUCTION

In MATLAB, features such as configuration of the low-level numeric representation of variables, assignment of specific data types and dynamic type specialization are directly supported by the language. In simulation tasks, such features provide significant support for developers that need to explore non-uniform fixed-point representations, monitor specific variables throughout a timing window and include handlers to observe specific behaviors. However, these features are also extremely cumbersome, error-prone and tedious to use. Each time they are used, developers are forced to perform invasive changes on the base code, namely adding "temporary" new code that must subsequently be removed before delivering the final version.

As MATLAB is typically used as a specification rather than an implementation language, tools to translate MATLAB code to the target programming language are important to achieve high-levels of productivity and efficiency. However, in order to attain a given desired efficiency level, the high abstraction level provided by MATLAB requires that tools be guided by users. In fact, the issues described above arise often in relation to automatic synthesis of MATLAB specifications to a software language [1][2] or a hardware description language [3]. In the steps for finding efficient implementations, users have to conduct customized profiling schemes, monitoring techniques, and data type and word-length exploration, mostly through the invasive insertion of new code.

In the past, multiple research efforts attempted to automate specific implementation issues. For instance, the transformation from floating- to fixed-point data types was conducted with some restrictions to MATLAB specifications [3][4]. However, it is usually claimed that the developer should have full control of the development process. Data type conversions are not trivial and usually require the intervention of the developers. Devising data types and word-lengths requires complex precision and accuracy analyses, which tend to become even more complex when considering customized data types and word-lengths. Typically, developers of MATLAB models rely on the default MATLAB data type (i.e., double precision floating-point) when often the full precision is not required. In the implementation stages of an embedded system, developers need to analyze the data type and word-length trade-offs in order to produce efficient solutions. Furthermore, when using customized word-lengths, developers

[1] MATLAB is a registered trademark of MathWorks Inc. http://www.mathworks.com

may also have to explicitly define type conversions and the resulting word-lengths for the operations dealing with those types, as the associated type conversion semantic rules are not built-in in the programming language[2]. Even when this process is assisted by tools, developers must explore and evaluate different data type representations, often by modifying the base MATLAB code and by simulating their effects on the precision of the system. In addition, according to the system being targeted, the same MATLAB code base may have different final implementations, forcing the developers to maintain different code bases pertaining to the same core functionality.

Since MATLAB is commonly used as a modeling/specification language, most MATLAB code aims to maximize its scope of applicability and dynamic configurability but at the price of unacceptable overheads for most common embedded computing implementations. In addition, developers often need to evaluate multiple algorithm implementations for a specific function, and each of those evaluations requires modifications to the MATLAB base code. An approach to provide different MATLAB versions from the same input MATLAB code would provide an important mechanism to avoid cumbersome, error-prone, code conversions and the maintenance of different MATLAB versions.

The root cause of the aforementioned problems is lack of modularity for dealing with *secondary concerns*. Ideally, core functionalities would be represented in the MATLAB code while secondary concerns, such as verification of the type and the number of the arguments for a given call of a function, should be kept in separate *aspect modules* as proposed in the context of aspect-oriented programming (AOP) [5][6]. Such concerns are not supposed to be implemented in the target embedded system and the developer needs to remove (or unplug) them in the process of translating the MATLAB code to the target programming language. Modularization of such concerns allows us to keep the core MATLAB code ready for translation to the target programming language, and thus to be implemented in the target embedded system and to automatically generate the generic MATLAB code used as a model. The generation of this generic MATLAB code is the responsibility of an *aspect weaver* – a tool or compiler component that composes aspect modules to the other parts of the system.

Hence, an approach is needed to generate and maintain a clean version of a MATLAB code. By clean (or unpolluted), we mean a version of the MATLAB code that includes just the core functionality, necessary for the system implementation. Clean code does not include configuration code or secondary concerns needed only during development, which should be kept in separate aspect modules.

Specialization is important when implementing the MATLAB application in the embedded system. This specialization can be leveraged by an AOP approach to MATLAB. The removal of the aforementioned secondary concerns can be approached as a specialization. Examples include data type specialization – assigning non-default data types to variables – and array size and shape specialization – defining statically the size and the shape of an array instead of including MATLAB code to get those parameters dynamically.

In this article, we propose aspect-oriented extensions to MATLAB to assist developers in system modeling and exploration of specific features related to embedded systems' implementations. Our approach, which builds on our previous work [7], relies on the principle of separation of concerns [8][9][10] to separately handle data types and behaviors. One of the ensuing advantages is that a single version of the specification (i.e., a MATLAB code base) can be used throughout the entire development cycle without the need for maintaining multiple versions – as is the case with existing technology. This separation facilitates the development, simulation, exploration and implementation phases. The extensions we propose can be used in other languages as well, namely "MATLAB clones" such as GNU Octave[3] [11] and Scilab [12]. Furthermore, preliminary studies show that the use of aspect modules in the MATLAB context improves the quality of MATLAB programs [13].

The main contributions of this article are:

- An aspect-oriented language to extend MATLAB models with features for supporting aspect modules that separately enclose concerns related to specialization, configuration, and monitoring.
- An aspect-oriented approach to enable programmers to flexibly explore a range of data type specializations, which includes aspect-oriented rules to specify the semantic rules for data type conversions and word-length assignments in operations using different, possibly customized, data types.
- An approach designed to tackle the problem of managing multiple implementations that depend on the target system, without forgoing a "clean" version of the MATLAB high-level model.

The rest of the paper is organized as follows. Section 2 provides a short introduction to the MATLAB programming language. Section 3 describes the main motivation for our work. Section 4 presents the approach and describes the domain-specific aspect language. In Section 5 we present a number of test cases performed. Section 6 compares our approach to related work. Concluding remarks are presented in Section 7.

2. THE MATLAB PROGRAMMING LANGUAGE

MATLAB is a dynamic, interpreted, imperative programming language mainly based on array data types and operations on those types. It is widely used in scientific computing, control systems, signal processing, image processing, system engineering, simulation, etc. Mathworks – the company that developed and holds the language's rights – provides a complete integrated environment to develop MATLAB projects. The environment includes a number of suitable debugging features. It includes Simulink, a visual, component-based environment suitable for simulation of discrete and continuous systems. Several toolboxes (packages) are available 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. Over 1,500 books[4] dedicated to MATLAB attest to its wide adoption.

[2] Even if they are part of the programming language, it is an advantage to make possible the exploration of those rules, as for a specific implementation, we may prefer to reduce accuracy and to use less costly operations and word-lengths.

[3] GNU Octave: http://www.gnu.org/software/octave/.

[4] http://www.mathworks.com/support/books/

Like most interpreted languages (e.g., Perl and Python), 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 supported numeric data types include integers (with 8, 16, 32 and 64 bits) and single precision floating-point numbers. MATLAB supports other numeric representations by using specific toolboxes. They enable the assignment of specific 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 number and types of arguments. Dynamic type specialization enables variables to represent different data types during runtime. For instance, developers can simulate the same code by applying stimulus with different data types.

MATLAB[5] provides a number of features suitable for fast modeling such as the vast set of supporting packages (toolboxes), the Mathworks simulation environment (including also Simulink) and the expressiveness of the language as regards specifying operations on array variables (e.g., matrix manipulations and operations).

MATLAB is a *dynamic language*, i.e., variables are not explicitly declared and data types of the elements, size, and shape of array variables are dynamically defined based on the runtime context. By default, all data types are N-arrays using double precision (64 bits) data types. Arrays may have different dimensions, forming the shape of the array variable. Arrays can store a single element, vectors of elements, and matrixes of elements. Additional data types supported by MATLAB (see Figure 1) include arrays with heterogeneous elements (known as *cells*), structures, strings, booleans, and function-handlers. In addition to the double-precision floating-point data type, MATLAB also supports single-precision data types and integer (signed or unsigned) representations (with 8, 16, 32, and 64 bits).

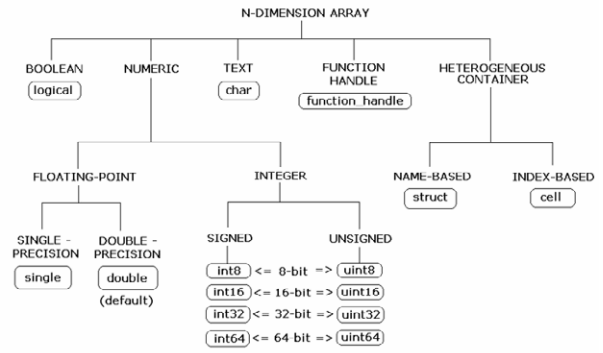

Figure 1. MATLAB data types (known in MATLAB as classes). (source: Mathworks1)

A MATLAB program comprises functions (known as M-files) and scripts. Functions have a name, arguments, and may have zero or more return variables. Functions can be called without passing all the arguments. Semantically, in MATLAB all function arguments are passed by value. To save memory, MATLAB execution environments pass by value only those arguments that a function modifies and by reference all the others. Scripts correspond to files with MATLAB code and without specifying inputs/outputs. Scripts can be also called in other scripts and in functions.

With the exception of the variables declared as *global*, MATLAB variables have local scope (function or script). A particular name (identifier) may refer to a variable or to a script or a function (a sub-function, a private function, or a function in the search path). In some cases, a MATLAB execution environment or compiler may have to postpone name resolution to runtime.

MATLAB also allows the definition of new classes. MATLAB classes can fall either in value classes or in handle classes. Both include two sections: *properties* (where attributes are declared) and *methods* (where functions are located). Handle classes also include an additional section named *events*. In this paper we do not consider features relating to classes as most MATLAB code is not developed according to the object-oriented paradigm.

Figure 2 shows a MATLAB function that receives as argument an N-array identified as *x* and returns an N-array identified as *y*. It represents an algorithm to perform the *Discrete Fourier Transform* (DFT) – a function widely used in signal processing systems. The statement in line 3 creates an array identified as *y* with the size of the array *x* and with all elements equal to 0 – represented in double-precision. This statement is not strictly needed as MATLAB dynamically reallocates memory space to store the elements of *y* during runtime, *i.e.*, during the assignments in line 9. The statement in line 6 creates an array named *t* with *N* values starting with 0 and equally spaced by *1/N*. The *for* loop in lines 8-9 computes and assigns a value to each *k* element of *y*, considering values of *k* from 1 to *N*. The calculations performed in the expression serving as the argument to the function *sum* in the loop body (line 9) are matrix-based calculations.

```
1. function [y] = dft(x)
2.
3.   y=zeros(size(x));
4.   N=length(x);
5.
6.   t=(0:N-1)/N;
7.
8.   for k=1:N
9.     y(k) = sum(x.*exp(-j*2*pi*(k-1)*t));
```

Figure 2. Simple MATLAB example (function to perform a Discrete Fourier Transform, source: [1]) – original code.

Figure 3 illustrates in a first example a function to add two variables *x*, *y*. Depending on the input variables, the sum operation performs an addition of two numbers or additions of the elements of two arrays. Thus, the result can be a simple number or an array of numbers resulting from the addition of the elements of the *x* and *y* arrays in equivalent positions. Figure 3 shows two further examples to add two variables. Note that the three examples may accomplish the same result but that depends on the shapes of the variables *x* and *y*.

MATLAB does not distinguish between accesses to variables and function invocations, e.g., *sin(1)* can be an invocation to the *sin* function with 1 as argument or an access to the first element of the array variable *sin*. When there is an assignment to *sin*, i.e., a statement defining *sin* as a variable (e.g., *sin=2;*), before the access *sin(1)*, a plain reference to *sin* represents the access to the first element of the array variable *sin*. In some cases, we can distinguish between the two by performing static name resolutions. Otherwise, a name can only be resolved dynamically.

[5] http://www.mathworks.com/products/matlab/

```
1.  % first example
2.  function [z] = add(x, y)
3.    z = x + y;
```

```
1.  % second example:
2.  function [z] = add(x, y)
3.   for k=1:length(x)
4.     z(k) = x(k) + y(k);
5.   end
```

```
1.      % third example:
2.      function [z] = add(x, y)
3.        [nr,nc]=size(x);
4.
5.        for k=1:nr
6.         for j=1:nc
7.           z(k,j) = x(k,j) + y(k,j);
8.         end
9.        end
```

Figure 3. Three different possibilities to add two variables.

Figure 4 illustrates an example that uses function handlers (denoted by @) where an argument may refer to a function or to a variable. Function handlers can be also used to define functions embedded in the MATLAB code as in the following MATLAB statement: *sqr = @(x) x.^2;*. In this case, one can use *sqr* as the function to calculate the square of each element in the array variable passed as argument.

```
1. …
2.  a = @sin;
3.  z1 = f1(0, a); % "x" argument in function "f1"
                   will refer the "sin" function
4.  …
5.  z2 = f1(1); % "x" argument in function "f1"
                   will refer a variable
6.  …
7.   function [z] = f1(y, x)
8.     if(y == 1)
9.       x = 3;
10.    end
11.    z = x(1);
```

Figure 4. Example where an identifier refers to an array variable or to a function depending on the path taken.

3. MOTIVATION

Uses of MATLAB in embedded systems range from the modeling of specific system components to the development of entire applications. Usually, developers start by modeling the core parts of the application in MATLAB, leveraging its high-level matrix-oriented and data type abstractions. The MATLAB environment is next used to simulate those core parts of the applications. The simulation may involve the interface of the cores to models of other system components or to real data traces previously provided. When work on the MATLAB model is over, developers need to translate the MATLAB code (or the parts to be mapped to the target system) to a native programming language (such as C). Although many efforts were carried out to automate this process, there is usually the need to allow developers to evaluate different specializations (e.g., of data types). These evaluations are much easier to do by simulation than by hand-coding. For efficiently supporting specialization, it is important to monitor variables and to evaluate different data type assignments. At the end of this process, an optimized MATLAB code version is used for implementation. Approaches that assist in this process comprise important contributions to increases in productivity.

There are many cases relating to the presence of code associated to secondary concerns that needs to be removed before the final implementation. This code can be encapsulated in aspect modules that can be composed through an aspect weaver. Multiple versions of the core, or base, code can be generated, e.g., one with the secondary code, another with type specialization for the final implementation. Our analysis of crosscutting concerns over MATLAB code repositories [14] discusses the intensive use of MATLAB built-in functions and variables responsible for acquiring runtime information about the size of arrays, verification of arguments of a function, data type conversions, etc. In a previously study [14] we conclude that the most common MATLAB secondary functionalities include:

- Messages and monitoring: messages to the user, warnings, errors, graphics visualization, monitoring, etc.;
- I/O data: reading data from file, writing data to file, saving an image, loading an image, etc.;
- Verification of function arguments and return values: default shapes and values for the arguments that may not be passed in certain function calls;
- Data type verification and specialization: check whether a variable is of certain type, configuring the assignment of data types to variables, etc.
- System: code that verifies certain system environment properties, to pause execution, etc.
- Memory allocation/deallocation: The use of the 'zeros' function is most of times used to allocate a specific array size. This avoids the reallocation for each new item to be stored in an array. Use of the 'clear' instruction that appears in some MATLAB functions is another example.
- Parallelization: use of parallel primitives such as 'parfor';
- Design space exploration: code to explore different specializations, different algorithms to solve the same problem, to find the number of iterations needed (e.g., to be above a certain precision).
- Dynamic properties: constructing inline function objects (inline), executing a string containing MATLAB expressions ('eval'), etc.

Secondary functionalities such as monitoring, data type specializations, and configuration features have an important role when developing models and code for embedded systems. Based on that, we centered our approach on those secondary functionalities without, however, loosing its applicability to other ones. The following subsections briefly describe monitoring, data type, and configuration concerns.

3.1 Monitoring

Monitoring can be used during the analysis phase enabling developers to acquire additional information to the typical information provided by profiling tools. Monitoring specific aspects of a MATLAB program can be important to analyze behaviors, value ranges of variables, number of accesses to matrix elements, control-flow paths taken during program execution, to acquire the accuracies when using specific data types, etc. Some monitoring aspects can be also important to acquire more information when dealing with legacy MATLAB code. An analysis of the resulting behaviors and execution characteristics can be of paramount importance to optimize code, to attain efficient implementations, and even to detect coding errors. However, sophisticated tracing, logging, and monitoring need some sort of code injection.

The monitoring capabilities assist developers in verifying the MATLAB models and to enrich profiling with customized information. The two following examples, presented in Figure 5 and Figure 6, illustrate monitoring for the two previous contexts.

Figure 5 presents an example taken from a MATLAB model of a PID (Proportional Integral Derivative) controller previously used in [15], for monitoring by plotting signal variables (*y* and *ref*) and displaying parameters (*kc*, *ti*, *td*, *ts*). This monitoring code was used by the developer when designing the model for the PID controller. It helped him/her to verify the behavior of the controller.

```
1.…
2.  figure;
3.  plot(t,y);
4.  hold on;
5.  plot(t, ref, '--');
6.  xlabel ('time (s)');
7.  grid;
8.  clear all;
9.  …
10.  disp(['kc = ', num2str(kc_ini)]);
11.  disp(['ti = ', num2str(ti_ini)]);
12.  disp(['td = ', num2str(td_ini)]);
13.  disp(['ts = ', num2str(ts_ini)]);
14.  …
```

Figure 5. Example of MATLAB code for monitoring.

For example, computing the range (minimum and maximum values) of variables in MATLAB code can be important to acquire information that can be used to define the word-lengths of variables. Figure 6 shows an example for monitoring the range for variables *top* and *bottom* in the MATLAB code in Figure 6(a). As illustrated in Figure 6(b), a call to the *range_find* function is inserted after each assignment to the target variable.

(a)
```
...
top = coef(j-1)*left-coef(j)*right;
bottom = coef(j-1)*right+coef(j)*left;
...
```

(b)
```
...
top = coef(j-1)*left-coef(j)*right;
range_find(top,'latnrm.top');
bottom = coef(j-1)*right+coef(j)*left;
range_find(bottom, 'latnrm.bottom');
...
```

Figure 6. Monitoring range values: (a) original MATLAB code; (b) MATLAB code after code insertion for monitoring value ranges (inserted code in italic).

3.2 Data Types and Shapes

As a default, MATLAB uses a matrix-oriented floating-point number representation with double precision. Thus, we need to specialize data types and shapes to avoid the runtime overhead of generic implementations. MATLAB does not support the notion of a scalar variable, just arrays of a single element. Representing this information in aspect modules may be important to complement techniques for type and shape inference, allowing a flexible, developer-guided, translation to native programming languages as well as yielding more efficient solutions – both in terms of code size and execution time.

Some implementation requirements entail the use of case-specific bit-widths to represent numeric data (integer and real numbers) so as to obtain the required accuracy. Non-standard bit-widths can be exploited, e.g., to save resources and to speedup performance through specialized and lower latency arithmetic operators [16] or through sub-word level parallelism [17]. In several digital signal-processing systems, the use of fixed-point arithmetic is common practice, due to the resulting efficient support, e.g., when targeting *Digital Signal Processors* (DSPs) devoid of hardware floating-point units or specific hardware. Examples include implementations based on *Field-Programmable Gate Arrays* (FPGA). Specific architectures may also use specialized data types (e.g., floating point arithmetic over data types not defined by the IEEE 754 standard). Such implementations require multiple tests to identify the bit-widths that yield the required accuracy (acceptable quantization errors) and behavior. Several authors proposed methods to automatically translate floating- to fixed-point representations (see, e.g., [3-4], [18-21]). Some methods rely on profiling, while others rely on static schemes. Although this is an important topic, the translation usually serves just to assist the designer, since no method is fully automatic and none can be applied without restrictions. In certain cases, designer experience and knowledge of the system requirements (which may go beyond accuracy, e.g., relating to dynamic range or precision) is the key factor to the success of the final implementation. Therefore, refinement of simulation and specification play an important role at both the data and the behavioral levels.

The MATLAB environment includes special packages to manage fixed-point representations. MATLAB provides two toolboxes for fixed-point computations: *Filter Design Toolbox* and *Fixed-Point Toolbox*. *Filter Design* provides functions (*quantizer* and *quantize*) to quantize values represented as, e.g., doubles in fixed-point representations. *Fixed-Point* provides 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 require specific changes in the code to be implemented. Such changes are error-prone, tedious, and difficult to maintain. In many cases, producing each new instance of the space being explored requires manual adaptations of large sections of code. Typical changes include insertion of statements, addition of function arguments, configuration of different data types, and definitions of global variables. Often, the developer must manage multiple versions of the same core specification.

To illustrate, consider as an example the MATLAB code from Figure 2. A simple MATLAB script to test it is shown in Figure 7(a). To use uniform fixed-point data types in the test, we merely add a line of MATLAB to the test program - see Figure 7(b). In this test, we use *fi(x, 1, 9, 5)*, which constructs a numeric fixed-point object with value *x*, signed representation, 9-bit word length, and 5-bit fraction length. However, to test the function using fixed-point representations with specialization of *every* variable and operation, we need to modify the original function, as shown in Figure 8. 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.

(a)
```
x=[1 2 3 4 5 6 7 8]; % input values
dft(x);
...
```

(b)
```
x=[1 2 3 4 5 6 7 8];

% fi(v,s,w,f) returns a fixed-point object
% with value v, signed property value s,
% word length w, and fraction length f
x=fi(x, 1, 9, 5); % new statement.

dft(x);
...
```

Figure 7. Example of a MATLAB script – Test of *dft* function: (a) with double-precision data types; (b) with a uniform fixed-point representation.

During the design phase, we usually need models that closely resemble implementation details. As an example, it may be necessary for computation results to be with fixed-point numeric representations, to validate the final implementation using a comparison between *Hardware Description Language* (HDL) and MATLAB simulations. Modeling with specialized fixed-point representations is important because such implementations are usually needed to satisfy requirements such as low power dissipation, low energy consumption, better performance and fewer hardware resources.

Note that this kind of data type specialization is also needed when object-oriented programming is used. Even if specific built-in class support for fixed-point data types is used, there is always the need to directly specify the required data type specializations. Though we do not focus on the object-oriented case, we believe our approach can be used in that context as well. Testing that hypothesis is left for future work.

```
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 8. Simple MATLAB example – code needed to model specialized fixed-point bit-widths.

3.3 Configuration Features

The configuration features targeted by our approach often arises when multiple, different implementations of a given function must be stored and managed. As an example, consider arithmetic division that can be implemented with look-up tables, iterative algorithms, or a combinatorial divisor. Each implementation may affect the overall system accuracy differently. Such systems require modeling prior to implementation, which entails changes in the original code giving rise to multiple versions of the code. Configuration features ameliorate the management problem. For instance, it would be helpful if developers could specify the implementation used by a simulation in a given development phase without changing the base MATLAB code. Using the modularity provided by an AOP approach, at a specific development stage, developers can opt to the use of a certain algorithm implementation by specifying that option through an aspect.

4. ASPECT-ORIENTED APPROACH

Our approach envisages the usage of two separate groups of source files to model a given system: (1) MATLAB code representing the primary behavior and (2) rules written in our aspect-oriented language. Aspect-oriented rules are mainly used to (a) reassign data types to variables in the MATLAB code, (b) introduce handlers and monitoring features, and (c) configure a function with a given implementation. A rule aims to facilitate development of embedded systems that require refinement of specific features required for implementation of the original specification. The proposed rules have declarative semantics as opposed to the imperative semantics of MATLAB. In our language, an *aspect* encloses one or several such rules, i.e., one rule is part of an aspect module. The proposed rules can be divided in the following groups according to their semantics:

- **Monitor/logging rules** help users observe the runtime characteristics of MATLAB variables. They include special behavior related to monitoring, such as returning the maximum value of a specific variable during the simulation period.
- **Handler rules** are a kind of assertions that ensure certain conditions hold during the simulation period.
- **Type assignment rules** are used to bind different types to the variables of the MATLAB specification, to specify type semantic rules in expressions, and to deal with name resolution.
- **Configuration rules** are used to statically bind a different implementation to a given function or operator.

The above rules are proposed based on observed needs while developing and implementing real embedded systems. The rules are represented by new specific constructs, e.g., to specify type assignments and to insert MATLAB code segments in specific locations. Usually, these segments add behavior rather than modifying existing functionality and can be, e.g., instructions for data allocation or display. Concerns from other categories may be proposed, but are out of scope of the present article. Note, however, that although we focused our current approach on the support to the rules previously described, the approach can be used to represent other concerns that can be expressed through insertion of sections of code.

Figure 9 presents the outline of the proposed system. As referred previously, aspect-oriented rules and MATLAB code are specified in different, separate files. A transformation engine – the *aspect weaver* – is responsible for generating new MATLAB code that includes the features composed according to the rules added to the system. The current implementation of the weaver uses the MATLAB compiler framework previously presented in [2]. The framework makes use of a strategic programming approach – Tom [22][23] in this case – to transform the intermediate representation (IR) of the input MATLAB code. The weaving process

presented in this paper is static, i.e., applied at compile time and consists of three stages. A first stage, named *mat2tir*, is responsible for transforming MATLAB input code into a Tom-based IR (*tir*) [22][23]. A middle stage named *tir2tir* modifies the input *tir* into according to the aspect rules fed into the system at this stage. Finally, a stage named *tir2mat* produces a MATLAB representation of the modified *tir*.

A library of MATLAB functions is fed into the system, comprising custom MATLAB functions that may be used in aspect rules. Although we consider here a MATLAB to MATLAB weaver, our approach can also be used to control the translation of MATLAB to other programming languages such as C, or hardware description languages such as VHDL or Verilog. An example is the current C code generation support provided by a stage, named as *tir2c* [2].

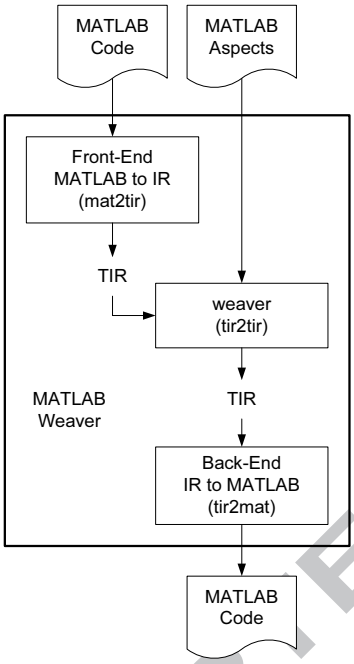

Figure 9: Outline of the MATLAB-based system enhanced with aspect-oriented rules.

4.1 The Aspect-Oriented Language

The aspect-oriented language supports the specification of join points by means of specific patterns of code, code locations and specific events traceable to specific code constructions, such as function calls and accesses to array elements. An excerpt of the grammar of the language is presented as an appendix to this article. According to the classification proposed by Stein et al. [24], the conceptual model for join point selections is based on control flows.

The pattern language includes a number of keywords to identify some properties (see Table I). Some of those keywords just require simple pattern matching at the source code level. However, to identify calls to functions in the MATLAB code, a dynamic analysis must be performed to distinguish between function calls and array accesses. We use a set of simple rules and delegate such distinctions to the runtime identification ([25] uses a similar approach). The resolution is based on the following rules:

- Assignments to identifiers require those identifiers to refer to array variables or functions if the identifier in the right hand side is preceded by '@';
- Use of identifiers is considered to refer to variables if there are only definitions (assignments to the variable, declaration of that variable as global) of that identifier reaching that particular use. In the case of multiple definitions without name resolution reaching an use, the identifier may refer to a function or a variable depending on the specific branches taken.
- Uses of identifiers followed by '(' … ')', where non-integer constants and strings are used as arguments, always refer a function call.

Figure 10 illustrates the use of the property *key* to identify specific MATLAB keywords in the code. They are specified as a set of keywords: *{for, while, if, end, else}*. The aspect rule presented in Figure 10 drives the weaver to insert *fprintf* statements for each keyword of the set in the code being analyzed. Those *fprintf* statements will trace blocks of MATLAB code during runtime.

Table I. The properties currently considered.

Property	Description
<var>	Identify variables in the MATLAB code
<call>	Identify function calls in the MATLAB code
<function>	Identify functions in the MATLAB code
<tag>	Identify tags (e.g., @label) in the MATLAB code using MATLAB line comments (they start with %)
<program>	Identify all the MATLAB files of a given program
<key>	Identify MATLAB reserved keywords in the code

The join point model supported by the language was designed with the aforementioned tasks in view (e.g., monitoring, logging, handling, type-assignment and configuration). Its constructs – call, function, key, program, type and var – are static in nature. Nevertheless that does not preclude future work extending them so that they can be used as clauses constraining the set of join points captured the other part of a pointcut expression. This would be similar to what happens to, e.g., the *args* clause of AspectJ, which can be used both to specify argument types and to constrain the join points to just those whose number of arguments match the *args* clause. In this paper, the clauses are used in "static mode" only.

Our aspect-oriented language uses the identifier (*id*) attribute to pinpoint where each join point occurred. Figure 10 shows an example of an aspect rule using indexing. This rule allows the weaver to insert code to trace the code blocks where each program execution passes (this can be helpful for debugging purposes and for detecting and localizing faults). The statement in line 2 of Figure 10 produces as many *key* elements as the number of join point shadows related to the elements in the join point selectors in the set `{for, while, if, end, else, elsif}`. The use of the attribute *id* (line 3) allows the weaver to insert a distinct number corresponding to each occurred join point shadow. This is helpful to represent the particular join point. For instance, Figure 11(a) shows an aspect rule which results in the insertion of MATLAB code to count the number of times the execution of the program passes in the body of each FOR-type loop in the code. The attribute *id* (lines 3 and 4) is used in this aspect to allow the inclusion of a different variable for each join point of type *for* (line 2). Figure 11(b) shows a simple example highlighting the code resulting from the weaving process when this aspect rule is applied.

Other attributes being currently used are the *get* and *set* attributes of the *var* join point selector. They can be used to select uses (reads) or definitions (writes) of variables, respectively. Extending the support of other attributes will increase the potential of the language to express customized monitoring rules.

```
1. rule trace_blocks
2.   foreach key in {for, while, if, end, else,
3.      elsif} do
4.    insert.before:
         fprintf(1,'block %d\n',<key.id>);
5.   end
6. end % trace_blocks
```

Figure 10. Aspect rule to insert code to trace executed code blocks.

```
1. rule count_passes_in_each_for
2.   foreach key in {for} do
3.    insert.before:
4.                        count_<key.id> = 0;
5.    insert.after:
         count_<key.id> = count_<key.id>+1;
6.   end
7. end % count_passes_in_each_for
```

(a)

```
1. count_738 = 0;
2. for k=1:nr
3.   count_738 = count_738+1;
4.   count_654 = 0;
5.   for j=1:nc
6.     count_654 = count_654+1;
7.     z(k,j) = x(k,j) + y(k,j);
8.   end
9. end
```

(b)

Figure 11. (a) aspect rule to insert code to count executed code blocks; (b) example of MATLAB code showing (in italic) the code statements inserted according to the aspect rule.

Figure 12(a) shows an aspect rule that results in the insertion of MATLAB code to count the number of times the execution of the program passes in two specific locations defined with the tags "@lab1" and "@lab2" in the code. As the counting must take into account the initialization of each variable, the rule uses the dependent constructions to specify the locations of each initialization ("@lab3" and "@lab4"). This example also includes the insertion of two statements to print the information related to the values stored in each counter variable. These two print statements are placed after the tags "@lab5" and "@lab6".

```
1. rule count_passes
2.  foreach tag t1 in {@lab1, @lab2} do
3.   insert.after:
4.     count_<t1> = count_<t1>+1;
5.   dependent:
6.     foreach tag t2 in {@lab3, @lab4}
7.       insert.after:
8.         count_<t1> = 0;
9.     end
10.    foreach tag t3 in {@lab5, @lab6}
11.     insert.after:
12.       fprintf(1,'count: %d\n',count_<t1>);
13.    end
14.  end
15. end
16.end % count_passes
```

(a)

```
1. %@lab2
2. count_lab1 = 0;
3.
4. for k=1:nr
5.   %@lab1
6.   for j=1:nc
7.     count_lab1 = count_lab1+1;
8.     z(k,j) = x(k,j) + y(k,j);
9.   end
10.end
11.%@lab5
12.fprintf(1,'count: %d\n',count_lab1);
```

(b)

Figure 12. (a) aspect rule to insert code to count the number of times the execution passes in two different locations; (b) example of MATLAB code (with tags) showing in italic the code statements inserted according to the aspect rule.

As illustrated in the example of Figure 12(b), a different variable is used for each count and each variable is initialized, incremented, and reported in three different locations. To preserve the names of the variables in each of these locations, we need to use the *dependent* construction and `tag` identifiers (Figure 12(a)). Note, however, that this aspect rule can be replaced by two different aspect rules (each one similar to the aspect rule illustrated in Figure 11(a)), one per counter variable. However, the expressiveness of the synthetic mechanism proposed by the aspect rule in Figure 12(b) would be lost.

4.2 Aspect-Oriented Rules

Type assignment and configuration comprise one of the most important categories of functionality for the aspect modules from the proposed approach. Using MATLAB, users start with a specification using double precision floating-point data types (the default MATLAB numeric data type).

To illustrate, the following MATLAB code represents a multiplication of two variables, previously assigned to constants. All the assigned and calculated values of this example are represented as doubles.

```
a = ...;
b = ...;
...
c = a*b;
```

Note that in MATLAB an operation using two *int16* type operands produces a result represented by a *int16* type. If we wish to test the code with integer data types, e.g. of 16 bits, the original code must be changed to the following:

```
a = int16(...);
b = int16(...);
```

```
...
c = a*b;
```

Using our approach the original code is kept as it is and we only need to add an assignment rule in an aspect file:

```
foreach var in program do set type=int16; end
```

This rule provides the transformation engine with the code needed to assign the type *int16* for each variable from the original code. In case we need to simulate the original code using different data types for each variable, we just need to use the rules below.

```
foreach var in program:{a,b} do set type=int16; end
foreach var in program:{c} do set type=int32; end
```

In this case we are specifying the following MATLAB code:

```
a = int16(...);
b = int16(...);
...
c = int32(a)* int32(b);
```

In such cases, applying aspect-oriented rules may entail the decomposition of arithmetic expressions into sub-expressions in order to apply different rules to each sub-expression. Suppose we have the following statement in a MATLAB specification:

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

To bind different specialized fixed-point representations to the sub-expressions computed by this statement, we 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 step is relatively straightforward, it requires changes in the original code, making it significantly less legible – and possibly less general. To address this problem, we provide 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:

```
with statement="a = b*c + d;" do
 decompose { v1=b*c; v2=v1+d; a=v2;};
end
```

This way, we can now provide type assignment rules to each variable (*a*, *b*, *c*, *d*, *v1* and *v2*). Note that the statements within brackets in the decompose command must be correct MATLAB code 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 range from monitors to the data sent to a file during a write to the values of specific variables during simulations. For instance:

```
foreach var in program do
  insert.after: print(file:'<var>.dat', <var>);
end
```

In some cases, we may want to register the maximum and minimum values assigned to a variable, e.g., when exploring bit-width analysis. This exposes the number of bits needed to represent certain values. Adding this behavior to the original code may require the use of global variables and the insertion of specific code to compute the maximum and minimum values for each assignment. This behavior is usually needed only during the development stages and must be latter removed. With our aspect-oriented rules, such behavior is kept separate from the original MATLAB code and rendered (un)pluggable. For instance:

```
foreach var in program do
  insert.after: print(screen, <var>:'max');
end
```

Handler rules can also help developers to expose the occurrence of specific values in variables. Example:

```
if func1:a>100
  print(screen, 'warning: value of func1:a exceeds:
',100);
end
```

Note that handler rules are similar to assertions.

Finally, configuration rules are used to assign a different implementation to an (arithmetic or logical) operator or a function. For instance:

```
with func1=f1 use configuration my_f1; end
```

Using the example from Figure 2 as a basis, Figure 13 illustrates a rule 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 14 shows an example of a rule to bind each operand of the "dft" function with a specialized fixed-point representation according to the result shown in Figure 8. Note that expressions already decomposed in the original code do not need decomposition commands in the aspect-oriented rules.

```
rule assignment1
  typedef fixed1 = fixed<1, 10, 5>;

  foreach var in function=dft do
   set type=fixed1;
  end
end
```

Figure 13. Quantification rule applied to the function presented in Figure 2 for uniform fixed-point representation.

Variables can be identified by their name preceded by the name of the function as in *func1:a* for variable *a* or as *func1:{a,b}* for variables *a* and *b*. The *with* construct obviates the use of the function name. It is illustrated in the example *with function=dft do* in Figure 14.

```
rule assignment2
  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>;
  typedef fixed9 = fixed<1, 32, 16>;

  with function=dft do
   with statement=
     "y(k)=sum(x.*exp(-j*2*pi*(k-1)*t));" do
   decompose %{
     v1=(k-1)*t;
     v2=pi*v1;
     v3=-j*2*v2;
     v4=exp(v3);
     v5= x.*v4;
     y(k)=sum(v5);
   }%;
   end

   foreach var in {t}  do set type=fixed1; end
   // or: with var=t do set type=fixed1; end
   foreach var in {pi} do set type=fixed2; end
   foreach var in {v1} do set type=fixed3; end
   foreach var in {v2} do set type=fixed4; end
   foreach var in {v3} do set type=fixed5; end
   foreach var in {v4} do set type=fixed6; end
   foreach var in {v5} do set type=fixed7; end
   foreach var in {x}  do set type=fixed8; end
   foreach var in {y}  do set type=fixed9; end
 end
end
```

Figure 14. Quantification rule applied to the function from Figure 2 for variable (specialized) fixed-point representation.

Each rule may have one or more commands. The commands for each aspect rule are executed in the sequential order in which they appear in the aspect. In the case of conflicts due to overlapping commands, the last command prevails. Figure 15 shows some examples of the proposed rules. To modify the ordering by which the rules are evaluated, one can use an *apply* statement (Figure 15, line 1), which allows a particular order to be specified.

The *Monitor1* rule in lines 3-20 of Figure 15 illustrates the monitoring. The rule starts by defining a set consisting of variables *a*, *b*, and *c* (line 4). The first *foreach* (lines 5-7) specifies the insertion of a print to the screen of the value of each variable in the set each time there is a write (attribute *set* in line 5) to that variable. The second *foreach* (lines 8-10) expresses an insertion of a print to the file "data.txt" of the value of each variable, each time there is a write to the variable. The last two *foreach* constructs (lines 14-16 and lines 17-19) specify the insertion of a print to the screen of the maximum and minimum values, respectively, of the variables, each time there is a read (attribute *get* in line 14 and line 17) of the variable.

Rule *assignment3* (lines 21-30) starts by defining the data type *fixed1* (line 22) which represents signed fixed-point values with 10 bits of word-length and having 4 bits of fraction. Line 23 defines the *real* data type as a single precision floating-point data type. The three foreach constructs of the rule express the assignment of all variables in the program as *real* (lines 24-25), all vars in the *module2* function as *fixed1* (lines 26-27), and all variables in *module3* function as *fixed1* (lines 28-29).

```
1. apply Monitor1; //several rules may be applied:
2.                 // apply Rule1:Rule2:Rule3;
3.
4. rule Monitor1
5.   set myVars1 = {a, b, c};
6.   foreach var.set in program do
7.     insert.after: print(screen, <var>:
8.                   'value for each change');
9.   end
10.  foreach var.set in myVars1 do
11.    insert.after: print(file:'data.txt', <var>);
12.  end
13.  foreach var.set in program do
14.    insert.after: print(screen,
15.                  <var>:'max');//mean, abs, etc.
16.  end
17.  foreach var.get in myVars1 do
18.    insert.after: print(screen, <var>:'min');
19.  end
20.  foreach var.get in module1 do
21.    insert.after: print(screen, <var>:'max');
22.  end
23.end
24.
25.rule assignment3
26.  typedef fixed1 = fixed<1, 10, 4>;
27.  set real=single;
28.  foreach var in program do
29.    set type=real;end
30.  foreach var in function=module2 do
31.    set type=fixed1; end
32.  foreach var in function=module3 do
33.    set type=fixed1; end
34.end
35.
36.rule handler1
37.  if func1:A > 100
38.    warning('value of A exceeds 100');
39.  end
40.end
41.
42.rule configuration1
43.  with function=f1, call=f2 use
44.    configuration f3; end
45.  with function=f2, operation="/" use
46.    configuration myDIV; end
47.end
```

Figure 15. Examples of aspect-oriented rules.

Rule *handler1* (lines 31-35) monitors variable *A* in function *func1* and specifies the output of a warning in case the value of *A* exceeds 100 (lines 32-34). Note that in the current version of the weaving, it is up to the user to apply this rule to MATLAB variables representing scalars (i.e., matrices of 1×1).

Rule *configuration1* (lines 36-41) shows two configuration actions. The first action (lines 37-38) specifies the use of function *f3* for the calls to *f2* from function *f1*. The second action (lines 39-40) specifies calling function *myDIV* for the division operators ("/") from *f2*. Although the current accepted configuration rules are simple, they are helpful when one needs to evaluate different implementations of specific functionalities, be they represented as functions or operators.

4.3 Type Conversion Rules for Expressions

Usually, when using customized data types, it is the responsibility of the user to express the data types resulting from operations involving customized data types. This burden can be avoided by

using built-in data type conversion rules. However, it is important to allow subexpressions to specify specific data types, which gives users the option of using lower accuracy types when maximum accuracy is not needed. This is common when using fixed-point customized data types as the word-lenghts needed to preserve accuracy may impose a large software and/or hardware overhead.

In the presence of expressions with more than one operation, one way to specify specific data types for the results of the subexpressions is to resort to expression decomposition and assign a specific data type to each subexpression that results from the decomposition. This is illustrated in Figure 14. This achieves the intended effect but entails modifying the expressions and references to auxiliary variables. It is, however, the scheme to apply when for some expressions one needs to specify particular data type assignments, not possible to address with the considered generic data type conversion rules.

An additional option is the specification of generic data type conversion rules. This is by default the preferable option. Our proposed aspect language includes a scheme to specify the semantic rules to be applied to each operation in expressions based on the data types presented in Table II. Note that the floating point data types only consider the single and the double precision representations specified by the IEEE Standard for Floating-Point Arithmetic (IEEE 754). The semantic rules expressions considered are of the following type:

```
"<operation> "<op>" <type> <id> ::= (<type> <id>)
[<type> <id>, <type> <id>]
```

The following statement is an example of the header of a semantic conversion rule to be associated to "+" operations involving a float and a fixed operand. It specifies that the resultant data type should be represented as a float:

```
operation "+" float f1 ::= (float f2) [float i1,
fixed f3]
```

The identifiers in the expressions are used in the semantic rules and the parameters (illustrated in Table II) can be used as fields of a given identifier as in the examples: `f3.w`, `i1.max`, and `f1.e`.

Table II. Data types and the corresponding parameters.

Types	Parameters	Description
int	w s	wordlength signed or unsigned
fixed	s w f	signed or unsigned wordlength fraction length
float	s m e	signed or unsigned mantissa length exponent length
all	max min	maximum value represented minimum value represented

Figure 16 illustrates a simple set of semantic rules to deal with a number of arithmetic operations. The main idea is to allow the developer to use semantic rules defined in the library of an aspect or to define and evaluate custom semantic rules. The rules include the possibility to specify commutative operations. This avoids the specification of all possible combinations having two different data types as operands. Figure 17(b) shows an example of applying the semantic rules presented in Figure 16 to the expression and data types shown in Figure 17(a).

```
rule semantic1

 commutative operation "*" float lhs ::=
            (float ir) [float op1, float op2]:
  if(lhs >= max(op1, op2))
    ir = max(op1, op2);//e.g., double if
                       //double ← double, single
  else
    ir = lhs; //e.g., single if
              //single ← double, single
  end
 end

 commutative operation "*" float lhs ::=
            (float ir) [int op1, float op2]:
   if (op2.max >= op1.max)
     if(lhs >= op2)
       ir = op2;
     else
       ir = lhs;
     end
   else
     if(lhs >= op1)
       ir = op1;
     else
       ir = lhs;
     end
   end
 end

 commutative operation "*" fixed f4 ::=
            (fixed f3) [fixed f1, fixed f2]:
  f3.s = f1.s OR f2.s;
  f3.w = max(f1.w, f2.w);
  f3.f = max(f1.f, f2.f);
 end

 % semantic rules for other operations
 % are omitted
end
```

Figure 16. Examples of semantic rules for data type conversions.

Figure 18 shows two different semantic rules for multiplications involving fixed-point data types. In the first case (Figure 18(a)) the multiplication of two operands in fixed-point representation does not loose precision because the resulting value is represented by a fixed-point representation with integer and fraction bit-widths given by the sums of the respectively bit-widths used in the input operands. In the second case (Figure 18(b)) the precision used to store intermediate results (i.e., results of the subexpressions in an expression) is the same as the precision used to store the result of the expression. With this approach, the developer can explore different semantic rules involving operations and data types. Note, however, that this approach assumes all arrays involved are *homogeneous* as regards element types.

Figure 19 illustrates the assignment of fixed-point data types to variables of the *dft* function presented in Figure 2 considering the use specific semantic rules (sentence *use semantic1;*) defined with our aspect-oriented language. In this case, the semantic rules are the ones partially specified in Figure 16. Note this is a distinct case from that presented in Figure 14, where a decomposition of the expressions "y(k)=sum(…);" is carried out to assign customized data types to each subexpression.

`// types specified`
`// by aspect rules`
`a: single;`
`b: int16;`
`c: single;`
`d: double;`
 	
`// expression`
`a=b*c*d;`
	`// resulting expression`
	`// with data types`
	`// defined by applying`
	`// semantic rules`
	`a = single(single(b*c)*d);`

(a)	(b)

Figure 17. Semantic rules example: (a) MATLAB code with an expression; (b) resulting code after applying semantic rules.

(a)
```
...
commutative operation "*" fixed f4 ::=
            (fixed f3) [fixed f1, fixed f2]:
  f3.s = f1.s OR f2.s;
  f3.w = f1.w + f2.w;
  f3.f = f1.f + f2.f;
end
...
```

(b)
```
...
commutative operation "*" fixed f4 ::=
            (fixed f3) [fixed f1, fixed f2]:
  f3 = f4;
end
...
```

Figure 18. Two examples of different semantic rules for fixed-point multiplications: (a) intermediate results with the precision required to store the result of the multiplication; (b) intermediate results using the precision used to store the result of the expression.

```
rule assignment7
   use semantic1;

  % "y(k)=sum(x.*exp(-j*2*pi*(k-1)*t));"

  set fixed = {overflow=wrap; round=floor};
  typedef fixed1 = fixed<1, 18, 16>;
  typedef fixed2 = fixed<1, 23, 20>;
  typedef fixed8 = fixed<1, 32, 16>;
  typedef fixed9 = fixed<1, 32, 16>;

  with function=dft do
   with var=t  do set type=fixed1; end
   with var=pi do set type=fixed2; end
   with var=x  do set type=fixed8; end
   with var=y  do set type=fixed9; end
  end
end
```

Figure 19. Quantification rule applied to the function from Figure 2 for variable (specialized) fixed-point representation using semantic rules defined with the aspect-oriented language.

The aspect-oriented extensions we propose also allow to "clean" MATLAB code by migrating code related to non-functional concerns (e.g., code to make a function more generic, code for monitoring, code to print results) to aspect rules. This yields less "polluted" MATLAB code and adds functionality required when using, e.g., a MATLAB to C compiler for mapping to an embedded system.

5. VALIDATING EXAMPLES

To validate our approach, we applied it to a number of MATLAB programs. We focused on the following five aspects:

- *Monitoring* for range value computation. This can be used to acquire the minimum word length of variables, e.g., the word length of the integer part of fixed-point representations;
- *Tracing* function calls, executed code blocks, number of iterations in loops, etc. This can be used to identify and locate software faults;
- *Counting* occurrences of specific operations, calls to a given function, number of times a variable is read or written, accounts of floating/point multiplication executions, etc.
- *Data type conversion*, e.g., to convert double precision to single precision or to fixed-point representation.
- *Exploration* of different configurations for a function. This can be used to evaluate trade-offs between implementation characteristics (e.g., precision vs. execution time).

In the following sections, we illustrate our approach with the following MATLAB codes: a program (*pid*) [15], three functions (*latnrm*, *fft*, *dft*) translated to MATLAB from codes taken from [26] and [27] and a repository with 26 MATLAB functions – *mat2c*[6].

5.1 Monitoring and Instrumenting

Profiling is an important task for optimizing applications. In addition to the computation of the percentage of overall execution time spent per function in the application (obtained by current profiling tools such as *gprof* or *profiling*), there are many other situations in which profiling is useful. For instance, finding the range for each variable in the program, as used by tools previously proposed [18]. Here, we use our MATLAB aspect-oriented approach to insert monitoring points that compute the range (minimum and maximum values) of variables in MATLAB code.

With a simple aspect description such as that shown in Figure 20, it is possible to specify the monitoring of the range of each variable in a program run. Applying this rule to the code of *latnrm* (32^{nd}-order Normalized Lattice filter processing 64 points) – see Figure 6(a) – yields the code shown in Figure 6(b). Figure 21 shows the trace obtained after weaving and running the function *latnrm*.

```
rule range_finder
  foreach var in program do
   insert.after: range_find(<var>,
                 '<function>.<var>');
  end
  with function=main do
   insert.before.end: report_range_find();
  end
end % range_finder
```

Figure 20. Aspect rule to insert code for determining in runtime the range values for each variable in the code.

[6] A copy of the MAT2C benchmarks, previously existent at http://www.ece.northwestern.edu/cpdc/pjoisha/MAT2C/, can be downloaded from: https://svn.strategoxt.org/repos/octave/octave-xmpl/mat2c-benchmark/

```
==== report range values of 13 variables:
var latnrm.data [min, max]: [1.136, 6.322]
var latnrm.coefficient [min, max]: [-0.81, 0.93]
var latnrm.internal_state [min, max]: [-1.12162,
3.35092]
var latnrm.NPOINTS [min, max]: [64, 64]
var latnrm.ORDER [min, max]: [32, 32]
var latnrm.bottom [min, max]: [-1.12162, 3.35092]
var latnrm.i [min, max]: [1, 64]
var latnrm.top [min, max]: [-2.04024, 6.322]
var latnrm.j [min, max]: [1, 32]
var latnrm.left [min, max]: [-2.04024, 6.322]
var latnrm.right [min, max]: [-1.12162, 3.35092]
var latnrm.sum [min, max]: [-0.101099, 3.87368]
var latnrm.outa [min, max]: [0.184243, 3.74967]
```

Figure 21. Report results of range values obtained for each variable in function *latnrm* after executing the woven code.

Table III presents the metrics of a number of examples in which the proposed approach was applied. It includes (1.) the number of *join point shadows* [28] – points in the source code that relate to a join point during program execution – when considering the aspect related to the insertion of code for "range finder", (2.) and (3.) the number of lines of MATLAB code (LOC), respectively, before and after the weaving, and (4.) the number of variables monitored in each of the examples. Line (6.) in Table III shows the *reduction in bloat due to tangling* – a metric proposed by Kiczales et al. [28] and called *aspectual bloat* in this paper. It compares the AOP and non-AOP versions of a system, using equation (1). It measures the degree to which the aspects are more concisely coded in the AOP-based implementation than in a non-AOP based implementation. Any number greater than 1 indicates a positive outcome of applying AOP. *Aspect code* is the code inserted due to aspects.

$$aspectual\ bloat = \frac{LOC(code\ after\ weaving) - LOC(original\ code)}{LOC(aspect\ code)} \quad (1)$$

The *Tangling ratio* metric (7.) was proposed by Lopes [29] and is based on the idea that the parts of the code associated to crosscutting concerns are "shadowed"[7]. The metric counts the *transition points*, i.e. the points in the source code where there is a transition from a shadowed area to a non-shadowed area and vice-versa. Tangling ratio is calculated using equation (2).

$$tangling\ ratio = \frac{\#\ transition\ points\ between\ aspect\ code\ and\ original\ code}{LOC(original\ code)} \quad (2)$$

Tangling ratio gives an idea of both the relative efforts a developer may need to add the code to the application and of the "code pollution" degree. Values for this metric start from 0 (no tangling) and have no theoretical upper bound. A value above 1 means there are more than one transition point per LOC on average.

The aspect module used for this experiment is the one presented in Figure 20, which has 8 lines of code. The results are presented in the "range finder aspect" section of Table III. For these experiments we have an increase of about 74% of lines of MATLAB code on average. With individual increases from 19 to 31, 24 to 43, 56 to 100, 506 to 892 for *latnrm*, *dft*, *fft*, and for *mat2c* repository, respectively. These experiments show an average of 24.5 join point shadows per MATLAB function. This is a clear indicator of the pollution degree and work effort that a simple instrumenting concern may originate. In our approach, this is achieved by automatic aspect weaving that avoids invasive changes on the original, core MATLAB code which is kept as it was.

Table III also shows the *aspectual bloat* (1), which ranges from 1.50 to 48.25 for these examples. These values also support claim that our approach brings benefits. The *aspectual bloat* is high when considering the MATLAB code of *mat2c* and *pid*. The reason is that one aspect is applied to more extensive MATLAB code. In fact, the *aspectual bloat* of the *mat2c* represents the effect of reusing the same code over multiple MATLAB functions.

Finally, the *tangling ratio* (2) ranges from 1.50 to 2.05 for these examples and aspects and once again strongly suggests there are benefits from using our approach. Note that *tangling ratio* values near or above 1 indicate the insertion of almost one secondary concern *in each line* of MATLAB code.

In a second monitoring example, we consider the report of the number of accesses to each variable. Figure 22 shows the resulting output after executing the *latnrm* MATLAB code woven with aspect code to determine the number of accesses (read or write) to each variable in the original *latnrm* code. In this case, the weaver identified 36 join point shadows (1.8× more than for the previous range *find* example to which 20 join point shadows were identified), which result in an even more "polluted" MATLAB code.

In a third example, we consider the report of the class of identifiers used in the MATLAB code. The analysis needs to deal with the case of MATLAB identifiers corresponding to multiple classes (e.g., an array and a function). This report can be important in MATLAB applications to acquire the identifiers corresponding to functions, classes, structs, and their types. For instance, this dynamic analysis may guide compilers or additional weaving with respect to name resolution. We show in Figure 23 the report output after executing the woven code of the *latnrm* example.

Table III. Results of applying aspects for a number of benchmarks (#LOC represents the number of lines of effective code statements).

	range finder				data type assignment	
MATLAB code / Metric	**latnrm**	**dft**	**fft**	**mat2c reposit.**	**l atnrm**	**pid**
1. #join point shadows	20	19	44	386	19	89
2. #LOC before weav.	19	24	56	506	19	268
3. #LOC after weav.	31	43	100	892	33	519
4. #vars monitored	13	13	30	254	n/a	n/a
5. #functions affected	1	1	1	26	1	13
6 #transition points	39	37	86	760	56	119
7. aspectual bloat	1.50	2.38	5.50	48.25	2.33	5.98
8. tangling ratio	2.05	1.54	1.55	1.50	2.95	0.44

[7] Note there is no relation between this code "shadowing" and the notion of joint point shadow.

```
==== report accesses of 11 variables:
var latnrm.bottom : 4033 accesses
var latnrm.i : 128 accesses
var latnrm.data : 64 accesses
var latnrm.top : 4096 accesses
var latnrm.j : 8064 accesses
var latnrm.left : 5952 accesses
var latnrm.right : 5952 accesses
var latnrm.internal_state : 6144 accesses
var latnrm.coefficient : 9984 accesses
var latnrm.sum : 4224 accesses
var latnrm.outa : 64 accesses
```

Figure 22. Report results of variable accesses obtained for each variable in function *latnrm* after executing the woven code.

```
==== report classes of 13 variables:
var latnrm.data : class char
var latnrm.internal_state : class double
var latnrm.NPOINTS : class double
var latnrm.ORDER : class double
var latnrm.bottom : class double
var latnrm.i : class double
var latnrm.top : class double
var latnrm.j : class double
var latnrm.left : class double
var latnrm.right : class double
var latnrm.coefficient : class double
var latnrm.sum : class double
var latnrm.outa : class double
```

Figure 23. Report results of classes obtained for each variable in function *latnrm* after executing the woven code.

5.2 Data Type Specialization

Regarding data type conversions, we show two examples where we have explored the fixed-point representations, specialized or uniform. The "data type assignment" section of Table III presents the results. The examples include the previous *latnmr* function and a MATLAB model of a PID (Proportional Integral Derivative) controller previously used in [15].

For the *latnrm* example, an *aspectual boat* of 2.33 and a *tangling ratio* of 2.95 confirm a positive outcome of our approach. In this example, there are transition points in almost every line of code. By coincidence, the number of join point shadows and the number of LOCs before weaving is the same (19). This is indicative of a highly polluted and difficult to read code.

The original MATLAB code of the *pid* has 268 lines of code. After weaving with the aspect rules defining fixed-point specialized data types a MATLAB code with 519 lines has been produced (1.93× more lines of code). The *aspectual bloat* is 5.98 and the *tangling ratio* is 0.44. The *tangling ratio* in this example is lower than 1 as most MATLAB code related to data type assignments is relatively well localized and thus yields a much lower ratio of transition points per line of code. Nevertheless the number of code modifications again suggests there are benefits in using the aspect-oriented approach.

5.3 Discussion

The previous examples illustrate some of the uses of the aspect-oriented language to extend MATLAB code with specific features as monitoring calls or data type specializations. In addition to the timing savings, the use of automatic features reduces the likelihood of manual code insertion errors. Thus, the proposed aspect features seem to provide valuable help to MATLAB programmers and system developers.

It is worth noting that even in the presence of some statements that appear just once in a function with many lines of code, there is justification for using the proposed approach in some cases. Through the aspect rules and aspect weaving, we acquire the option to generate multiple, case-specific configurations of a core code base. The verification of the number of arguments passed to a function is an example. These options are not implemented when translating the function to C code for the embedded target system. There are also cases where the code output by the aspect weaver has fewer lines of code but is "polluted" with calls to type conversions.

Although MATLAB was extended to support classes and objects, the use of these features remains almost totally absent from typical MATLAB systems. That is what can be concluded from the code repositories we have analyzed [14] and from the MATLAB industrial applications to which we had access.

Although our approach provides users with helpful mechanisms for monitoring, type assignment and configuration, it can be further extended in a number of ways:

- Providing extensions to deal with control-flow aware monitoring schemes. For example, one may need to specify monitoring actions dependent on particular sequences of function calls;
- Providing analysis of aspect rules in terms of conflicts that may exist between rules. For example, there might be more than one type assignment to the same variables and this reassignment may be intentional or accidental. Identifying those reassignments can help users;
- Extending the current simple statements accepted for code insertion, and thus avoiding in most cases the use of explicit target language code and the %{ and }% tags as in the approach in [42]. This will provide a more neutral approach as the code to be inserted can be specified in a language that is then translated to the target language by the weaving process.

Although our approach has been used in the context of MATLAB, it is also applicable to "MATLAB clones" such as Octave [11] and Scilab [12]. However, further analysis on this topic is required to assess how adequate to those "MATLAB clones" is our approach. This may call for more target-independent constructs to deal with possible mismatches between the various languages – possibly by using mapping rules.

We also believe that the approach can be also used in the context of other imperative programming languages. In future, we intend to perform further studies to assess the applicability of this approach as regards monitoring and type assignment so that it can be used on top of the LARA approach [42].

6. RELATED WORK

Most aspect-oriented approaches target general-purpose software programming languages, such as Java and C/C++, often in the context of general-purpose applications [9]. However, the specifics of embedded systems, regarding specific implementation properties and programming models, provide new use cases for aspect-oriented programming. Previous uses of AOP for debugging, instrumentation and monitoring retain their importance in the development for embedded systems. Other uses of AOP – such as type specialization – acquire greater importance in embedded systems. Below we describe the approaches related to that proposed in this paper.

In [6], Irwin et al. present AML, a system for sparse matrix computation that deals with crosscutting concerns (such as execution time and data representation) using AOP principles [28]. 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 a completely separated way from the main functionality. Thus, readability and maintainability of the behavioral code is not negatively affected by non-functional concerns. The AML system seems to have brought satisfactory results, as the authors report that their code in AML has similar speed as a standard version, yet it is smaller and less complex. They propose a "data representation" aspect module that is relevant for our work. This aspect module defines 5 dimensions for representing data: element type, dimension, representation, ordering, and orientation. AML was first described as an aspect-oriented system but some authors do not consider it as such [30].

Mück et al. [31] present a design methodology, based in SystemC and aspects, which allows components of operating system to be implement in hardware platforms. To validate the methodology, the authors discuss the implementation of a task scheduler and an aspect program. Aspects are used for on-chip debugging and define the following debugging features: (1) Watched dumps the state of a component whenever it is modified; (2) Traced signalizes every operation execution; and (3) Profiled counts the number of clock cycles needed by the component for a given operation. This approach also adopts the idea of having two different specification parts (main functionality and aspects), but differs from ours in several issues, namely in the adopted language (SystemC vs. MATLAB) and the focus (debugging vs. development).

Other researchers also propose the combined use of model-driven and aspect-oriented principles, concepts and techniques targeted for the embedded field. One common theme found in several research works is the use of the model-driven approach compounded with aspect-oriented techniques to improve separation of concerns at earlier phases in the software life cycle – modeling in the case of the works by Linehan et al. [32][33], Gray et al. [34] and Oliveira et al. [35]. In the case of Oliveira et al., requirements as well as modeling are subject to this approach.

Linehan et al. [32][33] propose an approach specially targeted for generating verification purposes, permitting the development of hardware verification testbenches, which the authors claim is easier to maintain, adapt and reuse. Gray et al [34] discuss the use of the model-driven approach for generating quality-of-service (QoS) adaptation rules within the simulation and implementation of distributed real-time embedded systems. This approach creates high-level graphical models representing QoS adaptation policies. The models are specified in a domain-specific modeling language (AQML) that helps in the separation of common concerns of an embedded system through different modeling views. Their primary contribution is an aspect-oriented weaver that performs model transformations across higher level abstractions to separate policy decisions that were previously scattered and tangled across the model. Oliveira et al. [35] also present a method for design space exploration of embedded systems that uses model-driven engineering and aspect-oriented concepts. The authors claim that their method provides better reusability, complexity management, and design automation by exploiting both MDE and AOD approaches in the earliest stages of the life cycle, including requirements.

To the best of our knowledge, our approach – initially proposed in [7] – is one of the first approaches to consider aspect-oriented extensions to MATLAB, especially aspect-oriented rules to specify code injection and assignment of numeric data types to a MATLAB specification. Our proposal differs from [6] 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 *multiple* possible implementations of a given core MATLAB specification. It does that without changing the original code and without the need to manage multiple versions of the same underlying specification. Moreover, most of the proposed aspect modules would be unsuitable to embed in the original specification in the form of annotations. There are various reasons for that. First, that would result in less legible code and would be cause of various kinds of hurdles whenever the original code needs to evolve. Second, it would still entail managing more than one version of the MATLAB specification when different data types for a given variable need be explored. Third, some of the rules are intended to be applied *globally*, not just to specific functions. With our approach, explorations can be performed with the same base MATLAB specifications by simply employing different aspect-oriented rules. Our approach uses a declarative type of aspect semantics suitable to be applied both locally and globally.

More recently, AspectMatlab was proposed [25] as an approach to extend MATLAB with aspects. AspectMatlab does not consider type assignments. The design of AspectMatlab is instead geared to the support of scientific computing, which is typically computation-intensive. For this reason, the join points supported cover elements that play important roles in computing-intensive applications, namely array accesses and loops. Though our proposed language also supports advices over loop constructs, its focus is on simulation, monitoring, and data type exploration.

Hendren [36] proposed the addition of typing aspects to MATLAB. The approach is based on a new kind of uses statement – *atype* – that captures runtime type information of variables and verifies their types. This is a specific case of monitoring and instrumentation that can be controlled by a weaver as the one proposed in AspectMatlab [25] or the one proposed in this paper. As with AspectMatlab, the primary motivation for proposing typing aspects is performance: modern MATLAB systems support JIT compilers, which require type information to produce efficient code.

Complementing the work presented in this paper, we have recently proposed a domain specific aspect language to enrich MATLAB with *code transformations* [37]. Those code transformations can be used to implement the aspect rules given in the approach presented in this paper. However, that approach addresses additional code transformations that can be used to optimize the MATLAB applications while our approach provides specific support to the exploration of data types and configurations and to the monitoring of specific program artifacts.

Approaches to code transformations have been extensively proposed in recent years. Pattern matching transformations have been proposed by some authors. An example was given by Bodin et al. [38] as a way to allow the user to specify specific code transformations.

It can be argued that the aspect rules presented in our approach could be specified using code transformation tools such as TXL [39]. That approach would also need the specification of the MATLAB grammar as well as the rules presented here. Note, however, that by using a strategic programming approach at the

intermediate representation (we use Tom [22][23]), we isolate the compiler front-end and back-ends from the weaver and contribute to an extensible compiler framework in terms of compiler optimizations, code transformations and code generation. Nevertheless we believe there is no additional reason not to use TXL as the transformer and code insertion engine, e.g., by translating our aspect rules to TXL rules.

Our approach to data type specializations also promotes the use of *active libraries* [40] in the context of MATLAB. In this approach, MATLAB libraries can be delivered to a specific implementation by using aspect rules that automatically produce woven MATLAB code with the required specializations.

Our approach to data type specialization is also being used in the compiler framework to generate C code from MATLAB specifications [2]. Thus, the approach presented in this paper not only assists in the early development phases but the implementation phases as well, by providing data type and shape information for the subsequent code generation steps. As our kind analysis stage is not so powerful than recent analysis techniques applied to MATLAB [41], it may need more intervention from the user to resolve some MATLAB names. Future work is expected to integrate more advanced kind analysis techniques.

The development of LARA [42], a domain-specific aspect-oriented language, has been also inspired by some of our ideas proposed in the context of extending MATLAB with aspects. LARA has been designed to be as agnostic to the target language as possible – though its main application has been to C programs – and is a more complex language as it addresses many concerns, such as code instrumentation, compiler optimizations, mapping decisions, type and code specialization and design space exploration strategies. The AOP language proposed in this paper is distinct from LARA in a number of ways. It is focused on a narrower set of concerns than LARA, uses an imperative semantic while LARA uses both declarative and imperative semantics and it is focused on MATLAB, while LARA has been proposed for multiple languages. Being specially focused to a particular set of concerns makes the language easy to use and easy to support by tools. We have plans to generate LARA aspects from the aspect rules proposed in this paper.

7. CONCLUSIONS

This paper presents an approach to add aspect-oriented rules to MATLAB specifications to assist developers of embedded systems in the exploration of implementation features – namely numeric data type configurations. MATLAB core behavior and aspect-oriented rules (e.g., numeric data type assignments) are specified and maintained in separate modules. Our approach allows developers to insert MATLAB code that is helpful for debugging, monitoring, and exploring numeric data type representations without changing the original MATLAB code. With this approach, the core MATLAB specifications are kept free from code dependent on the implementation and target system/architecture.

Our approach allows users to explore multiple, different implementations of embedded systems based on MATLAB specifications. We are able to maintain a base MATLAB code and to achieve different specializations, code insertions to trace and to acquire dynamic properties, through the use of aspects. This certainly contributes to modularity and code maintenance. In addition, our approach can be used as a support to some advanced MATLAB type and shape inference analysis systems as the results of those analyses can be represented by aspect-oriented rules.

One of the difficulties we found is the lack of MATLAB code considering some of the secondary concerns such as the ones including customized data types and monitoring. Most MATLAB code found in repositories represent generic, target independent, models. The use of MATLAB models considering custom data types is more related to subsequent stages of the design cycle, e.g., for embedded systems products. It is understandable that those models may not be public. The monitoring concerns occur during the entire design cycle and most of them are concerns that typically are not present in the end.

Although the current version of our approach provides useful mechanisms to express monitoring and data type assignments, it can be enhanced by considering other types of aspect rules and more sophisticated patterns to express join point selections. Extensions to the support of parameters would make rules more reusable.

From the derived results, it is advantageous for our approach if other metrics are also considered. In future, other metrics that have no correlation (see the *aspectual bloat* and the *tangling ratio* values presented for *latnrm* and *pid* in the previous section) should be used as well.

Complementary work in progress includes studies about other aspect-oriented rules, a more powerful pattern language, and a tool to manage strategies (the possibility to apply different sequences of aspect rules). In addition, we expect that our ongoing work on aspects related to complementary information can help a MATLAB compiler to map more efficiently MATLAB computations and data structures to the target architecture. One interesting research avenue is the automatic extraction of secondary concerns from MATLAB code to aspect modules.

8. ACKNOWLEDGMENTS

This work was partially supported by FCT (Portuguese Science Foundation) under the project AMADEUS (POCTI, PTDC/EIA/70271/2006).

9. REFERENCES

[1] R. Allen, Compiling high-level languages to DSPs, IEEE Signal Processing Magazine 22, 3 (2005) 47-56.

[2] R. Nobre, J.M.P. Cardoso, P.C. Diniz, Leveraging type knowledge for efficient MATLAB to C translation, 15th Workshop on Compilers for Parallel Computing (CPC'10), Vienna, Austria, 2010.

[3] S. Roy, P. Banerjee, An algorithm for converting floating-point computations to fixed-point in MATLAB based FPGA design, 41st Annual Design Automation Conference (DAC'04), 2004, pp. 484-487.

[4] S. Roy, P. Banerjee, An algorithm for trading off quantization error with hardware resources for MATLAB-based FPGA design, IEEE Transactions on Computers 54, 7 (2005) 886-896.

[5] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.-M. Loingtier, J. Irwin, Aspect oriented programming, European Conference on Object-Oriented Programming (ECOOP'97), LNCS 1241, 1997, pp. 220-242.

[6] J. Irwin, J.-M. Loingtier, J. Gilbert, G. Kiczales, J. Lamping, A. Mendhekar, T. Shpeisman, Aspect-oriented programming of sparse matrix code, Int. Scientific Computing in Object-

Oriented Parallel Environments (ISCOPE'97), LNCS 1343, 1997, pp. 249-256.

[7] J. M. P. Cardoso, J. M. Fernandes, and M. Monteiro, Adding Aspect-Oriented Features to MATLAB, in SPLAT! 2006, Software Engineering Properties of Languages and Aspect Technologies, workshop affiliated with AOSD 2006, March 21, 2006. Bonn, Germany.

[8] D.L. Parnas, On the criteria to be used in decomposing systems into modules, Comm. ACM 15, 12 (1972) pp. 1053-1059.

[9] R. Filman, T. Elrad, S. Clarke, M. Aksit (eds), Aspect-Oriented Software Development, Addison-Wesley 2005.

[10] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr., N degrees of separation: multi-dimensional separation of concerns, 21st International Conference on Software Engineering (ICSE'99), ACM, New York, NY, USA, pp. 107-119.

[11] J.W. Eaton, D. Bateman, S. Hauberg, GNU Octave Manual Version 3, 2009.

[12] C. Bunks, J.P. Chancelier, F. Delebecque, C. Gomez, M. Goursat, R. Nikoukhah, S. Steer, Engineering and Scientific Computing with Scilab, Birkhäuser, 1999.

[13] P. Martins, P. Lopes, J.P. Fernandes, J. Saraiva, J.M.P. Cardoso, Program and aspect metrics for MATLAB, 12th International Conference on Computational Science and Applications (ICCSA 2012), LNCS 7336, 2012, Part IV, pp. 217–233.

[14] M. Monteiro, J.M.P. Cardoso, S. Posea, Identification and characterization of crosscutting concerns in MATLAB systems, Conference on Compilers, Programming Languages, Related Technologies and Applications (CoRTA 2010), Braga, Portugal, 9-10 September 2010..

[15] J. Lima, R. Menotti, J.M.P. Cardoso, E. Marques, A methodology to design FPGA-based PID controllers, IEEE International Conference on Systems, Man, and Cybernetics (SMC'06), 2006, pp. 2577-2583.

[16] S. Mahlke, R. Ravindran, M. Schlansker, R. Schreiber, and T. Sherwood, Bitwidth cognizant architecture synthesis of custom hardware accelerators, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Synthesis 20, 11 (2001):1355-1371.

[17] K. Scott, J. Davidson, Exploring the limits of sub-word level parallelism, 9th Int. Conference on Parallel Architectures and Compilation Techniques (PACT'00), 2000, pp. 81-91.

[18] M. L. Chang, S. Hauck, Précis: A user centric word-length optimization tool, IEEE Design and Test of Computers 22, 4 (2005):349-361.

[19] D-U. Lee, A. A. Gaffar, R. C. C. Cheung, O. Mencer, W. Luk, G. A. Constantinides, Accuracy guaranteed bit-width optimization, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 25, 10 (2006):1990-2000.

[20] A. Nayak, M. Haldar, A. Choudhary, P. Banerjee, Precision and error analysis of MATLAB applications during automated hardware synthesis for FPGAs, Conference on Design, Automation and Test in Europe (DATE'01), 2001, pp. 722-728.

[21] P. Banerjee, D. Bagchi, M. Haldar, A. Nayak, V. Kim, R. Uribe, Automatic conversion of floating-point MATLAB programs into fixed-point FPGA based hardware design, 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'03), pp. 263-264, 2003.

[22] J.-C. Bach, É. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, A. Reilles, TOM Manual, Version 2.7, May, 2009 (http://tom.loria.fr)

[23] E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, A. Reilles, Tom: Piggybacking rewriting on Java, 18th International Conference on Term Rewriting and Applications (RTA'07), Paris, France, LNCS 4533, 2007, pp. 36-47.

[24] D. Stein, S. Hanenberg, R. Unland. Expressing different conceptual models of join point selections in aspect-oriented design, 5th International Conference on Aspect-Oriented Software Development (AOSD '06), 2006, pp. 15-26.

[25] T. Aslam, J. Doherty, A. Dubrau, L. Hendren, AspectMatlab: An aspect-oriented scientific programming language, 9th International Conference on Aspect-Oriented Software Development (AOSD'10), 2010, pp. 181-192.

[26] UTDSP Benchmark Suite, http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html.

[27] W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, Numerical Recipes: The Art of Scientific Computing, 3rd ed., Cambridge University Press, New York, 2007.

[28] H. Masuhara, G. Kiczales, C. Dutchyn, A Compilation and optimization model for aspect-oriented programs, Compiler Construction (CC2003), LNCS 2622, 2003, pp. 46-60.

[29] C. V. Lopes, D: A language framework for distributed programming, PhD Thesis, College of Computer Science, Northeastern University, USA, 1997.

[30] C. V. Lopes, AOP: A historical perspective (what's in a name?), in: R.E. Filman, T. Elrad, S. Clarke, M. Akşit (Eds.), Aspect-Oriented Software Development, Addison-Wesley, 2005, pp. 97–122.

[31] T.R. Mück, A.A. Fröhlich, M. Gernoth, W. Schröder-Preikschat, Implementing OS components in hardware using AOP, SIGOPS Oper. Syst. Rev. 46, 1 (2012) 64-72.

[32] É. Linehan, E. O'Toole, S. Clarke, Model-driven automation for simulation-based functional verification, ACM Trans. Des. Autom. Electron. Syst. 17, 3 (2012) 31.

[33] É. Linehan, S. Clarke An aspect-oriented, model-driven approach to functional hardware verification, J. Syst. Archit. 58, 5 (2012), 195-208.

[34] J. Gray, S. Neema, T. Bapty, A. Gokhale, D.C. Schmidt, J. Zhang, Y. Lin, Concern separation for adaptive QoS modeling in distributed real-time embedded systems, in: L. Gomes, J.M. Fernandes (Eds.), Behavioral Modeling for Embedded Systems and Technologies: Applications for Design and Implementation, IGI Global, 2010, pp. 85-113.

[35] M.F.S. Oliveira, M.A. Wehrmeister, F.A. Nascimento, C.E. Pereira, F.R. Wagner, High-level design space exploration of embedded systems using the model-driven engineering and aspect-oriented design approaches, in: L. Gomes, J.M. Fernandes (Eds.), Behavioral Modeling for Embedded Systems and Technologies: Applications for Design and Implementation, IGI Global, 2010, pp.114-146.

[36] L. Hendren, Typing aspects for MATLAB, 6th Annual Workshop on Domain-Specific Aspect Languages (DSAL'11), 2011, pp. 13-18.

[37] J.M.P. Cardoso, P. Diniz, M.P. Monteiro, J.M. Fernandes, J. Saraiva, A domain-specific aspect language for transforming MATLAB programs, Domain-Specific Aspect Language Workshop (DSAL'2010), 2010.

[38] F. Bodin, Y. Mével, R. Quiniou, A user level program transformation tool, International Conference on Supercomputing, 1998, pp. 180-187.
[39] J.R. Cordy, The TXL source transformation language, Science of Computer Programming 61, 3 (2006) 190-210.
[40] K. Czarnecki, U. W. Eisenecker, R. Glück, D. Vandevoorde, T. L. Veldhuizen, Generative programming and active libraries, in: M. Jazayeri, R. Loos, D. R. Musser (Eds.), Selected Papers from the International Seminar on Generic Programming, Springer, London, 1998, pp. 25-39.
[41] J. Doherty, L. Hendren, S. Radpour, Kind analysis for MATLAB, ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '11), 2011, pp. 99-118.
[42] J.M.P. Cardoso, T. Carvalho, J.G.F. Coutinho, W. Luk, R. Nobre, P.C. Diniz, Z. Petrov, LARA: An aspect-oriented programming language for embedded systems, International Conference on Aspect-Oriented Software Development (AOSD'12), 2012, pp. 179-190.

10. APPENDIX

Below is an excerpt of the LL(1) grammar of the aspect-oriented language.

```
Start ::= (<APPLY> <IDENTIFIER> ( ":" <IDENTIFIER> )* ";" )* ( Rule )*
Rule ::= <RULE> <IDENTIFIER> ( Use )* ( Statement | InjectCode | TypeDef )* ( DataTypeConvRules )* <END>
InjectCode ::= <IF> IfRule (TargetCode)+ <END>
Set ::= <SET> ( ( <IDENTIFIER> "=" ( ( "{" <IDENTIFIER> ( "," <IDENTIFIER> )* "}" ) | Types | <IDENTIFIER> ) ) | … | ( <FIXED> "=" "{" FixedPointProperties "=" ValuesFixedPointProperties ( "," FixedPointProperties "=" ValuesFixedPointProperties )* "}" ) ) ";"
FixedPointProperties ::= <OVERFLOW> | <ROUND>
ValuesFixedPointProperties ::= <WRAP> | … | <FLOOR>
TypeDef ::= <TYPEDEF> <IDENTIFIER> "=" Types ";"
Use ::= <USE> <IDENTIFIER> ";"
Types ::= <INT8> | <INT16> |…| <DOUBLE> | <SINGLE> | Fixed | Float | Int
Fixed ::= <FIXED> ( "<" <NUMBER> "," <NUMBER> "," <NUMBER> ">" )?
Float ::= <FLOAT> ( "<" <NUMBER> "," <NUMBER> "," <NUMBER> ">" )?
Int ::= <INT> ( "<" <NUMBER> "," <NUMBER>)?
Statement ::= ForEachStatement | Insert | Set | WithStatement
ForEachStatement ::= ( <FOREACH> ( <KEY> | <TAG> | Var ) ( <IDENTIFIER> )? <IN> ( ( <IDENTIFIER> | ( ( <PROGRAM> | <FUNCTION> ) ( "=" SetOfJPs )? ) ( ":" SetOfJPs )? )? ( "{" JPList "}" )? ) <DO> ( ( Insert | Set ) ( <DEPENDENT> ":" ( DependentStmt )+ <END> )? )+ <END> )
Var ::= <VAR> (“.” (<SET> | <GET>))?
SetOfJPs ::= <IDENTIFIER> | ( "{" ( <IDENTIFIER> ( "," <IDENTIFIER> )* ) "}" )
WithStatement ::= <WITH> ( ( ( ( <FUNCTION> | <MODULE> ) "=" <IDENTIFIER> ) ( ( <DO> ( Statement ( <DEPENDENT> ":" ( DependentStmt )+ <END> )? )+ <END> ) | ( ( "," ( ( <CALL> "=" <IDENTIFIER> ) | ( <OPERATION> "=" OpsForConfigRules ) ) )? ( <USE> ( <CONFIGURATION> <IDENTIFIER> ";" )+ <END> ) ) ) ) | ( <STMT> "=" <STRING> <DO> Decompose <END> ) | ( Var "=" SetOfJPs <DO> ( Statement )+ <END> ) )
DependentStmt ::= <FOREACH> ( <KEY> | <TAG> | Var ) ( <IDENTIFIER> )? <IN> ( ( <IDENTIFIER> | ( ( <PROGRAM> | <FUNCTION> ) ( ":" <IDENTIFIER> )? ) ) | "{" JPList "}" ) <DO> ( ( Insert | Set )+ ) <END>
JPList ::= ( ( ( "@" )? <IDENTIFIER> ) | Keys) ( "," ( ( ( "@" )? <IDENTIFIER> ) | Keys ) )*
Insert ::= <INSERT> "." ( <BEFORE> | <AFTER> | <AROUND> ) ( "." ( <END> | <BEGIN> ) )? ":" ( TargetCode )+
Keys ::= <END> | <IF> | <ELSE> | <FOR>
TargetCode ::= <TARGETCODE>
| <FPRINTF> "(" Arg ( "," Arg )* ")" ";"
| <IDENTIFIER> ( ( Param "=" ( ( <IDENTIFIER> Param Op <NUMBER> ) ";" | ( <NUMBER> ) ";" ) ) | ( "(" ( Arg ( "," Arg )* )? ")" ";" ) )
| <PRINT> "(" ( <SCREEN> | ( <FILE> ":" ( ( <IDENTIFIER> "." <IDENTIFIER> ) | ( <QUOTED> ) ) ) ) "," Arg ( "," Arg )* ")" ";"
| <WARNING> "(" <QUOTED> ")" ";"
Decompose ::= <DECOMPOSE> <TARGETCODE> ";"
Op ::= "+" | ”-“ | … | “*“
Arg ::= <NUMBER>
| <IDENTIFIER> ( Param )?
| <QUOTED> ( Param ":" <QUOTED> )?
| Param ( ":" <QUOTED> )?
| "\"" Param ( ":" <QUOTED> )? "\""
```

Param	::=	"<" (<KEY> | <IDENTIFIER> | <VAR>) ("." <IDENTIFIER>)? ">"
DataTypeConvRules	::=	(<COMMUTATIVE>)? <OPERATION> OpsForConvRules TypesOnCOnvRules (<LHS> | <IDENTIFIER>) "::=" "(" TypesOnCOnvRules (<IR> | <IDENTIFIER>) ")" "[" TypesOnCOnvRules <IDENTIFIER> ("," TypesOnCOnvRules <IDENTIFIER>)? "]" ":" (IfConvRule | SimpleStatement)+ <END>
IfConvRule	::=	<IF> "(" (<LHS> | CompleteIdentifier) (">=" | "<=" | ">" | "<" | "==" | "!=") (CompleteIdentifier | BuiltInFunctions "(" (CompleteIdentifier ("," CompleteIdentifier)*)? ")") ")" (SimpleStatement | IfConvRule)+ (<ELSE> (SimpleStatement | IfConvRule)+)? <END>
SimpleStatement	::=	(<IR> "=" (<LHS> | BuiltInFunctions "(" (CompleteIdentifier ("," CompleteIdentifier)*)? ")" | CompleteIdentifier) ";")
	|	(CompleteIdentifier "=" (CompleteIdentifier | BuiltInFunctions "(" (CompleteIdentifier ("," CompleteIdentifier)*)? ")") (Op (CompleteIdentifier | BuiltInFunctions "(" (CompleteIdentifier ("," CompleteIdentifier)*)? ")"))? ";")
BuiltInFunctions	::=	<MAX> | … | <MIN>
OpsForConvRules	::=	<STRING>
OpsForConfigRules	::=	<STRING>
CompleteIdentifier	::=	<IDENTIFIER> ("." (<IDENTIFIER> | BuiltInFunctions))?
TypesOnCOnvRules	::=	(Types | <FLOAT> | <INT>)