Program and Aspect Metrics for MATLAB*

Pedro Martins[1], Paulo Lopes[1], João P. Fernandes[1,2], João Saraiva[1], and João M. P. Cardoso[2]

[1] HASLab / INESC TEC, Universidade do Minho, Portugal
{prmartins, plopes, jpaulo, jas}@di.uminho.pt
[2] Universidade do Porto, Faculdade de Engenharia, Departamento de Eng. Informatica, Porto, Portugal
{jmpc}@fe.up.pt

Abstract. In this paper we present the main concepts of a domain-specific aspect language for specifying cross-cutting concerns of MATLAB programs, together with a suite of metrics that is capable of assessing the overall advantage of introducing aspects in the development cycle of MATLAB software. We present the results of using our own suite to quantify the advantages of using aspect oriented programming, both in terms of programming effort and code quality. The results are promising and show a good potential for aspect oriented programming in MATLAB while our suite proves to be capable of analyzing the overall characteristics of MATLAB solutions and providing interesting results about them.

Keywords: Aspect Oriented Programming, Matlab, Aspect Metrics

1 Introduction

MATLAB [1] is a high-level, interpreted, mathematically-oriented domain-specific language which has some key characteristics such as being based on matrix data types, not requiring variables declaration and including operator overloading. Combined with function polymorphism and dynamic type specialization these features, together with the MATLAB environment provided by *MathWorks* [2], create an interesting environment to easily model and simulate complex systems.

The fact is that, in MATLAB as in most programming languages, tasks such as exploiting non-uniform fixed-point representations, monitoring certain variables or including handlers to observe specific behaviors are extremely cumbersome, error-prone and tedious tasks. Indeed, each time one of these features is necessary, invasive changes to the original MATLAB program need to be performed.

In addition, as MATLAB allows a higher-level of abstraction than, for example, the C programming language, the *de facto* standard embedded systems

* This work is funded by the ERDF through the Programme COMPETE and by the Portuguese Government through FCT - Foundation for Science and Technology, projects ref. PTDC/EIA/70271/2006 and PTDC/EIA-CCO/108995/2008. The first and third authors were also supported by FCT grants BI3-2011/PTDC/EIA/70271/2006 and SFRH/BPD/46987/2008, respectively.

programming language, aspect rules may also be used to specialize the MATLAB input program to different target architectures. These specializations may include data types, array shapes, and implementations of a given function, and also contribute to the deterioration of the overall quality of MATLAB code.

The goal of this paper is three fold: firstly, we provide software complexity metrics for MATLAB programs. These metrics, based on the Halsted's complexity metrics [3], provide a quantification of the overall quality of a MATLAB program. Secondly, we present a set of metrics for an aspect oriented extension of MATLAB, and thirdly, we also present aspect metrics to quantify the quality of aspect MATLAB programs. Finally, we present our experimental results of using both suites of metrics on real MATLAB code and aspect MATLAB programs. We show the results we obtained by applying our metrics and then we analyze the impact of using aspects. We also show, using concrete examples, how the quality of a MATLAB program can see significant increases when an aspects language is used.

In [4], the authors have suggested an aspects language, and a compiler for it, that is capable of concern modularization and supports specific scientific computation tasks. Being aware of this Aspects Oriented Programming (AOP) approach to MATLAB, we use our own aspects language because we are familiar with it, because it is simple to use and also because it supports the creation of aspect-oriented versions of MATLAB programs. In this paper we argue that MATLAB programming can suffer from concerns pollution and that introducing concerns separation makes programming easier, faster and can help producing code with more quality. Nevertheless, those improvements are orthogonal to any aspect oriented approach.

The remainder of this paper is organized as follows. In Section 2 we introduce the MATLAB programming environment and two practical examples. In Section 3 we introduce our aspect oriented approach, which we use to show how the examples of the previous section can be improved. In Section 4 we introduce our suite of metrics to assess the quality of both MATLAB programs and their aspectualized versions. In Section 5 we show the results of applying the metrics to a set of MATLAB programs, and in Section 7 we conclude this paper.

2 MATLAB

MATLAB is a high-level matrix-oriented programming language to implement computationally intensive tasks faster than with traditional programming languages. The code presented below is an excerpt of a MATLAB function that implements the Discrete Fourier Transform (function dft_custom).

Example of a MATLAB *Program:*

```
function [mag] = dft_custom(x, n)
twoPI = 2.0 * pi;
n2 = n/2;
for i = 1:n2
    xre(i) = x(i);
end
```

```
for i = 0:n-1
  arg = twoPI * i / n;
...
```

There are several characteristics to notice regarding MATLAB code. Firstly, to provide faster development, data types and shapes of variables are not specified. In the particular case of this example, from the assignment twoPI $= 2 * pi$, one could assume twoPI would be stored as a scalar of double-precision floating-point type[3]. Internally, twoPI will actually be stored as a single-element array of type double.

Secondly, array variable shapes are inferred during program execution. Indeed, the assignments to variables expose, at runtime, the shape of those variables. At a certain point of a MATLAB program, assigning arg = twoPI * i / n will imply that arg is a single-element array, while at another point of the code assigning arg=[1 2; 3 4; 5 6] will imply that arg now refers to a 2×2 matrix. While these dynamic features speed up program development, they complicate the translation of MATLAB to non-dynamic languages. For many systems, the overhead to implement this dynamic behavior is just not acceptable.

In addition to functions, MATLAB offers the possibility of structuring code in scripts, which very much resemble bash scripts in UNIX (a simple sequence of instructions), or even Object Oriented Programming (OOP), by defining classes and applying standard object-oriented design patterns which allow code reuse, inheritance, encapsulation, and reference behavior. Despite promising, the introduction of OOP in MATLAB is still very recent and has not yet seen a broad use, whereas functions are part of most existing MATLAB solutions.

Despite some syntactic similarities with C, semantically they are very different. For example, the variable whose value is returned by a function, mag in the case of the previous example, is declared in the function signature itself, instead of by a primitive such as return, as in C; also, MATLAB functions may explicitly return more than one value.

Finally, function calling in MATLAB is easy in the sense that all functions are polymorphic, so a programmer knows that whatever variable he/she applies to a function, it will always produce a concrete result. Take as an example the $\times$ operator, that can be used to multiply integers, floats and even matrices.

2.1 Tracing in MATLAB

Tracing is a specialized method to obtain execution information of a program, and it is a technique that is frequently used during the debugging of a program.

A concrete scenario where tracing would be of practical interest is the following: in case a function, such as dft_custom, is not producing the expected results, the programmer may want to print all values assigned to some variables when executing that function for a particular input. Observing all the printed values may then help in understanding and identifying the coding error.

In order to implement this tracing, one could follow the strategy of planting printing instructions throughout the original source code, every time an assignment occurs. For dft_custom, this strategy would result in the code shown next:

Example of a MATLAB *Program with tracing:*

[3] pi is a constant in MATLAB representing the value π.

```
function [mag] = dft_custom_tracing(x, n)
twoPI = 2.0 * pi;
disp('The value of twoPI is:'); disp(twoPI);
n2 = n/2;
disp('The value of n2 is:'); disp(n2);
for i = 1:n2
    xre(i) = x(i);
    disp('The value of xre(i) is:'); disp( xre(i) );
end
for i = 0:n-1
  arg = twoPI * i / n;
  disp('The value of arg is:'); disp( arg );
end
```

While this is a strategy that can be manually performed for small-sized programs, using it for large-sized applications can be a tedious and unproductive task. Moreover, the code for tracing must also be manually removed from the final version of the code. In Section 3, we show how tracing can be achieved, in an elegant and systematic way, using the notion of program aspects.

2.2 Specialization of MATLAB programs

The high level features of MATLAB make it a widely used language for fast development and for focusing on problem solving instead of on implementation issues. Type declarations, for example, do not exist in MATLAB.

The high level features, however, make it dificult to generate efficient implementation code from MATLAB programs. Also, and this is particularly true for embedded systems, it is commonly necessary to specify a particular MATLAB implementation to a target system, with hardware or computational restrains.

Let us consider the function dft_custom again. Transforming its code to a program in a different programming language used in target embedded systems can be challenging due to the dynamic nature of MATLAB's type system: type information is not explicit. To generate efficient code we do not only need type information, but we may also need different versions of the specialized function, one for each of the target systems. Next, we present the redefinition of dft_custom, where a single fixed point data type is explicitly defined by the programmer.

Example of a MATLAB *Program with specialization:*

```
function [mag] = dft_custom_specializing(x, n)
q = quantizer('fixed','floor', 'wrap', [32 16]);
twoPI = 2.0 * pi;
   twoPI = quantize(q, twoPI);
n2 = n/2;
   n2 = quantize(q, n2);
for i = 1:n2
  xre(i) = x(i);
  xre(i) = quantize( q, xre(i) );
end
for i = 0:n-1
  arg = twoPI * i / n;
  arg = quantize(q, arg);
end
```

In order to achieve this specialization, we created the MATLAB object q, defined using quantizer [5], which itself takes a series of arguments that determine the properties of q. The properties are used throughout the code to specialize the variables that are used to the data type whose properties are defined in q. In this case, we aim at targeting a generic system with variables in signed fixed-point mode (fixed), rounded towards negative infinity (floor), wrap on overflow (wrap) and has respectively 32 and 16 bits for the word length and for the fraction[4].

Regarding the code for specializing dft_custom that we presented above, it is as cumbersome as the code for tracing that we also present above. Indeed, it follows the same inefficient methodology that we have followed before. Again, the problem of data type and shape resolution and the generation of different implementations according to the target domain and architecture can be solved using aspect-oriented programming (AOP) [6].

As we discuss in the next section, our aspect language can be used to extend MATLAB programs with transformation and specialization rules that help the compiler to achieve more efficient code considering a certain target system. Furthermore, due to the modularity of our solution, the programs we obtain can easily be adapted to different deployment environments.

3 Aspect MATLAB

As mentioned in the previous section, the flexibility of the MATLAB language sometimes hinders performance and forces programmers to develop specialized versions of the same program. Furthermore, when it comes to evaluating specific features, such as tracing or including handlers to watch certain behaviors, the programmer is overwhelmed by cumbersome, error-prone and tedious tasks, which imply invasive code in the original MATLAB program. In [7], we have proposed aspect-oriented features to support triggering conditions and monitoring variable values, as well as a draft of an aspect language to support these features. In this paper, we briefly describe our aspect language specification, with primitives to create aspects, specify pointcut expressions, and apply transformations.

The original base program is free of language enhancements and sources remain legal MATLAB. The proposed Domain Specific Aspects Language (DSAL) enables programmers to retain the obvious advantages of a single source program representation while allowing the implementations to explore a wide range of specific solutions at reduced programming and maintenance costs.

The template shown below illustrates the structure of an aspect module; the same file can have various aspect modules.

The structure of an aspect module:

```
aspect aspect_name
  select: Join Point Capture
  apply: Action Description :: execute before | after | around
end
```

Each aspect has a main constructor: aspect, which initializes the aspect and gives it a name, and two main sections: select and apply, where the join point

[4] Besides these options, MATLAB offers a wide range of possibilities to specialize types, as shown in [5].

and the action are declared. The section apply is followed by a primitive execute where, similarly to AspectJ [8], we define if the alteration to the join point occurs before, after, or around the join point, replacing the original code.

Arrays	Variables/Constants	Functions
add( )	read( )	call( )
get( )	write( )	function( )
size( )	declare( )	head( )

Table 1: Primitives for join point capture.

The join points capture works in functions, variables and arrays, and the functions that capture join points are given in Table 1. In the next sections, we show how concrete aspects may be defined to achieve the same tracing and specialization results that we have presented in Section 2.

3.1 Tracing

In Section 2.1, we have shown how to manually change the original dft_custom function to perform tracing. This was achieved by inserting obtrusive instructions in the original function definition. In this section, we present how to concisely specify such features using our aspect language, as specified in the next aspect.

The structure of an aspect to implement tracing:

```
aspect variable_tracing()
  select: write()
  apply(): "disp('The value of"++name++"is: ');disp("++name++");"
  :: execute after
end
```

First, we define an aspect with name variable_tracing that is responsible for tracing a variable. The join point write detects when a value is written to a variable. The apply primitive inserts the new code and the execute primitive demands its insertion to be after the join point.

Aspects are automatically woven to the original code in order to create a new, valid MATLAB program that has tracing capabilities, as shown in Section 2.1. This means that the changes performed by the programmer are easy to realize in practice and that, after they are used, it is easy to discard them. In fact, the original MATLAB function remains untouched during the entire process.

3.2 Function Specialization

MATLAB types are not mandatory: the dynamic type system allows the programmer to create variables and functions without specifying their types. When targeting MATLAB into a specific target, however, type information is crucial not only to produce efficient code but also to make the solution compatible with different processor architectures or other hardware requisites.

Specifying generic functions is easily done with our aspect oriented language for MATLAB and is very similar to the aspects shown in the previous section for

tracing. The main difference here is that instead of only inserting information after the variables in order to force them to our custom types, we also have to define the quantizer object, that represents these types. Therefore, we now need two aspects: one that is executed only once, which is responsible for creating the quantizer object, and another that acts every time an assignment occurs, forcing that assignment into the desired type. These two aspects are presented next:

The structure of aspects to implement specialization:

```
aspect variable_specialization()
  select: head(dft_custom)
  apply(): "q = quantizer('fixed','floor', 'wrap', [32 16]);"
  :: execute after
end
aspect variable_specialization()
  select: write()
  apply(): name ++ "= quantize(q," ++ name ++ ");"
  :: execute after
end
```

The use of our aspect language makes it easier to specialize a MATLAB solution, but it also allows fast development and deployment of applications on heterogeneous environments where traditional programming techniques would not only be tedious and time-consuming but also prone to generate erros.

4 Metrics for MATLAB Programs

In the previous sections, we introduced both MATLAB and an aspect oriented extension for it. In this section, we briefly present complexity metrics for MATLAB and we introduce aspect MATLAB metrics. The idea is to use the complexity metrics to assess the quality of a MATLAB program and compare it to its aspect version.

4.1 Complexity Metrics for MATLAB

To assess the complexity of MATLAB programs, we use the *lines of code* metric and the *Halstead's* complexity metric suite [3].

Lines of Code (LOC): this metric is frequently used in software engineering to assess the quality of source code. Through it one can predict the effort needed to create the program and the cost of maintaining it after it is produced.

Halstead's Complexity: this suite of metrics was developed to measure a program's complexity directly from source code. The suite is composed by six measures that emphasize the complexity of a program: *Program Vocabulary, Program Length, Calculated Program Length, Volume, Difficulty, and Effort.*

Despite being easy to calculate, in order to automate the measuring process we have to define strong rules for identifying the operands and operators [9]. Let n_1, n_2, N_1, and N_2 be the number of distinct operators, the number of distinct operands, the total number of operators, and the total number of operands. The metrics that constitute the Halstead suite are, then, defined as follows:

- **Program Vocabulary (VOC)**: $n = n_1 + n_2$
- **Program Length (PL)** : $N = N_1 + N_2$

- **Calculated Program Length (CPL)**: $\hat{N} = n1 \times log_2 n_1 + n_2 \times log_2 n_2$
- **Volume**: $V = N \times log_2 n$
- **Difficulty**: $D = \frac{n_1}{2} \times \frac{N_2}{n_2}$
- **Effort**: $E = D \times V$

It is important to notice that these results provided by Halstead are not very useful by themselves, since they do not have units and therefore there is no qualitative interpretation for them (which is well-known problem of the Halstead suite). Consequently, we apply them in the basis that they are useful only when directly comparing different MATLAB sources.

4.2 Metrics for Aspects in MATLAB

Together with the metrics for MATLAB programs, we introduce a suite of metrics to help measuring the impact and the benefits of using an aspects oriented language when programming in MATLAB. For this we adapt the four software metrics introduced by [10] and [11] to MATLAB.

Concern Diffusion over Lines of Code (CDLOC): this metric counts the number of transition points, in the source, in and out of of zones where a concern starts and ends (shadowed zones). The use of this metric requires a shadowing process that partitions the code into shadowed areas and non-shadowed areas, being the code inside the shadowed areas lines of code that implements a concern. Transition points are points in the code where there is a transition from a non-shadowed to a shadowed area and vice-versa [11].

Tangling Ratio (TR): this metric gives an estimation about tangling on the program source code [10]. In the context of MATLAB, we can define it using the following formula:

$$Tangling\ Ratio = \frac{CDLOC\ program}{LOC\ program}$$

Concern Impact on LOC (CILOC): this metric gives us the ratio between a original MATLAB source code free of concerns, and the code after being transformed by our aspect language. This metric allows us to have a first intuition about the impact of using aspects in terms of lines of code, and it is given by the formula:

$$Concern\ Impact\ LOC = \frac{LOC\ of\ concerns\ free\ program}{LOC\ of\ transformed\ program}$$

The range of the results ranges from zero to one, where one means there are no concerns on the MATLAB solution and, therefore, there is no advantage in using an aspect oriented language. As we will see though, this is very uncommon since concerns are usually an important part of any software solution, being it MATLAB or not.

Aspectual Bloat (AB): measures the aspects in terms of LOC bloat in the MATLAB programs [10]. It is calculated by the following formula:

$$AspectualBloat = \frac{LOC\ with\ concerns - LOC\ without\ concerns}{LOC\ of\ aspects}$$

When the result of this metric is 1, it means that the number of lines written for the aspects plus the number of lines written on the MATLAB program without aspects is equal to the number of lines of the MATLAB program with aspect oriented language. With this result, it might seem that there is no advantage in using an aspects language, but even if the effort, in terms of lines of code was the same, the use of an aspects oriented language has other advantages, such as creating a program which is more modular and, consequently, easier to maintain and update.

5 Metrics Evaluation

With the metrics presented in the previous section, we extended our MATLAB front-end (which uses the MATLAB to Tom-IR tool [12]) in order to be able to apply the metrics to both the original MATLAB source code and its aspect oriented variants.

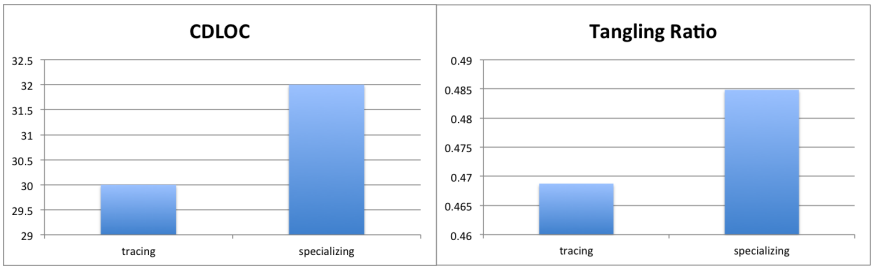

Figure 1: CDLOC and Tangling Ratio on the manually transformed versions of dft_custom.

5.1 Computing Metrics for Aspect MATLAB

In order to present our metrics we will use the examples provided in Section 2, consisting of both versions of function dft_custom: the one with variable tracing, as shown in Section 2.1 and the one with variable specialization, as shown in Section 2.2. Before, we had manually and intrusively written the code responsible for tracing and specialization. After that, we were able to run the first two metrics presented in Section 4.2, CDLOC and TR. These two metrics give us an indication of how many concerns exist in the source code and how much impact they have in the overall code quality. The results are presented in Figure 1.

We can see that both versions of the function dft_custom would benefit from the usage of an aspect language. Indeed, they have around thirty transition points in their code (as shown in the left chart of Figure 1), between a concern and the functional code of the application, and each transition point has the

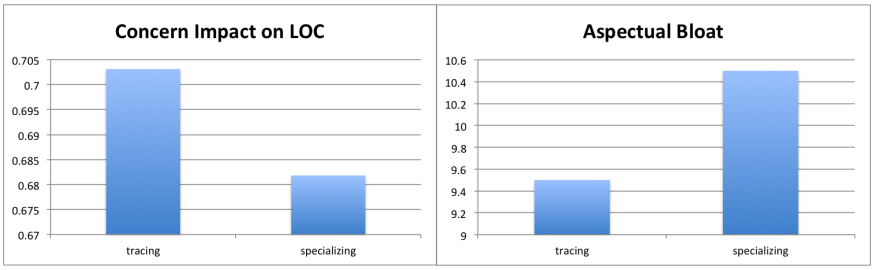

Figure 2: Concern Impact and Aspectual Bloat on the aspect-oriented versions of dft_custom.

potential to be transformed into an aspect. Here, as it is often the case, we do have various concerns, and therefore transition points handled by a single aspect makes their usage even more valuable. Regarding TR (right chart of Figure 1), both functions also show clear signs of a high degree of code tangling.

A significant amount of concerns in the source code makes it harder to understand and maintain. The fact that the metrics achieved such expressive results for dft_custom shows a good potential for aspects to be applied: they are particularly good in modularizing and aiding on implementing such features.

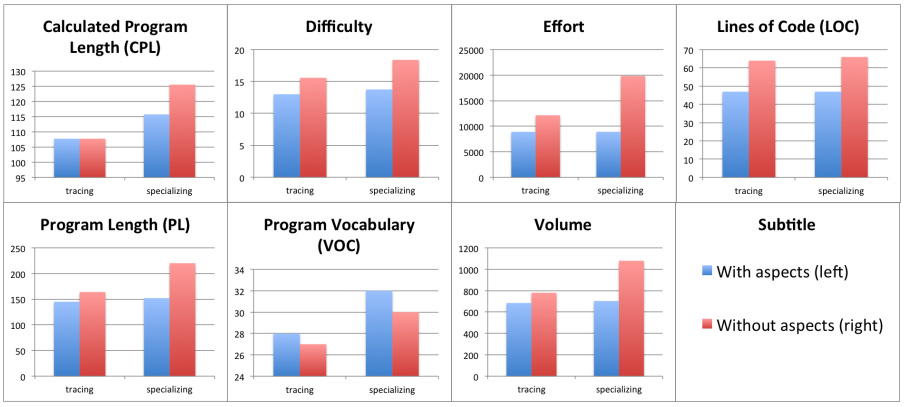

Figure 3: Metrics for dft_custom with tracing and specialization, implemented with and without our aspects language.

Our next step was to write, using our aspect language, modules that automatically generate the traced and specialized versions. These aspects are similar to the ones presented in Section 3, and make it not only simple and easy to obtain the different dft_custom functions, but also to backtrack any transformation in case we want to revert their application. In fact, the traditional method to

backtrack a manually transformed function is to manually remove all the code that implements the aspect functionality.

In Figure 2 we show, through the use of metrics CILOC and AB the actual effect that using AOP had on function dft_custom.

The first metric results, presented in the left chart of Figure 2, show the relation between the lines on a version of the code without concerns and on the same code after being transformed by our aspect language. This shows how less effort is needed by using an aspect language. In this case, and particularly on the case with tracing, the effect was noticeable: we were able to inject a significant number of lines of code only through the use of a single aspect.

The second metric results, shown in the right chart of Figure 2, seem promising too. This metric shows how much aspect code we had to write to change the original source code. The observed results indicate that the programming effort was greatly reduced by using aspects.

5.2 Computing MATLAB Metrics

So far, we have shown that the use of aspects can help on implementing new features while minimizing the traditional negative impact of extending source code. In this section we try to assess whether there are quality improvements in the original program, with added aspects, when compared to the manually transformed versions. For this, we compute the metrics presented in Section 4.1 on all versions of the dft_custom function, whose results are presented in Figure 3.

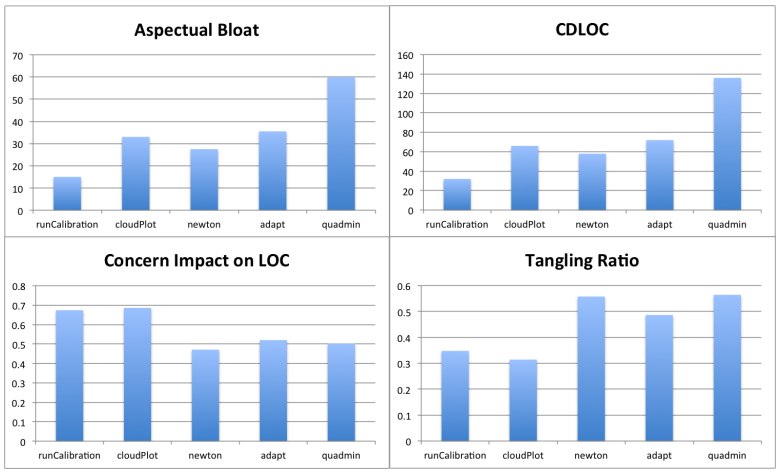

Figure 4: Aspect metrics in a set of five MATLAB functions.

This metrics suite seems to provide strong evidence that using aspects increases the overall code quality. The version with aspects is always clearer, shorter and easier to read. Some results actually show a large improvement, such as the metric *Difficulty*, that shows improvements in the order of 89% and

VOC, that shows gaps in the order of 88%. Other metrics show smaller advantages, such as CPL that shows improvements in the order of 8%. Still, the overall analysis of all results leads us to believe that using aspects in MATLAB programs can create programs with better quality.

5.3 Quality Analysis in Real-Life Applications

To further test the impact of our aspects language, we used a set of five functions taken randomly from MATLAB's File Exchange [13]. This website is a community-based repository for MATLAB functions, applications and scripts, where users can upload their implementations and download programs.

For each application, we took one source code file (very commonly, a MATLAB solution is made out of various source files) while being careful to pick applications from the groups with best rating or with the highest number of downloads or comments. We did so to ensure that our set is actually representative of the code usually found in MATLAB.

Next, we manually inserted new features in the code in order to specialize it and to trace its variables, similarly to the dft_custom examples presented in Section 2. We also implemented these features using our aspects language, i.e., with aspects that transform the MATLAB sources into new versions that support tracing and specialization.

In Figure 4 we show the aspect metrics applied to this set of functions. The results are similar to the ones found for dft_custom: using aspects reduces the effort of implementing specialized versions as seen, for example, on the tangled concerns on the code. In Figure 5 we show the metrics used for MATLAB quality assessment. Again, the overall results prove that using aspects decreases the size of the solution while keeping the code easier to maintain and reducing the effort to understanding it. Some metrics are particularly positive, with improvements of around 90%, from the traditional implementation to the aspects oriented version, as for PL on the quadmin function with tracing. The *Effort* metric applied to the same function also shows results on the order of 64%. The charts in Figure 5 confirm the overall promising results of our aspect oriented approach.

5.4 Quality Analysis of the Matlab program IMPACTED

In order to further conrm the results obtained so far, we have applied our metrics suite to the MATLAB program IMPACTED [14–16]. This program focuses on hazard avoidance techniques for controlled landings and has been the object of several studies. It represents a mature, highly complex solution composed by various functions and gives us a good environment for testing MATLAB code regarding both code quality and the potential for AOP introduction.

We have chosen a subset of this package, representing approximately 270 lines of code obtained through profiling, which gave us the hotspot in terms of the execution, represented by ten functions that represent the most important computations together and highest overheads. We did so because typically it is not necessary to separate concerns on all the elements of a solution. Some elements represent auxiliary functions, simpler and easier to control. The core computations, on the other hand, represent the core functionalities and therefore the parts where errors are more crucial, and where code control is harder.

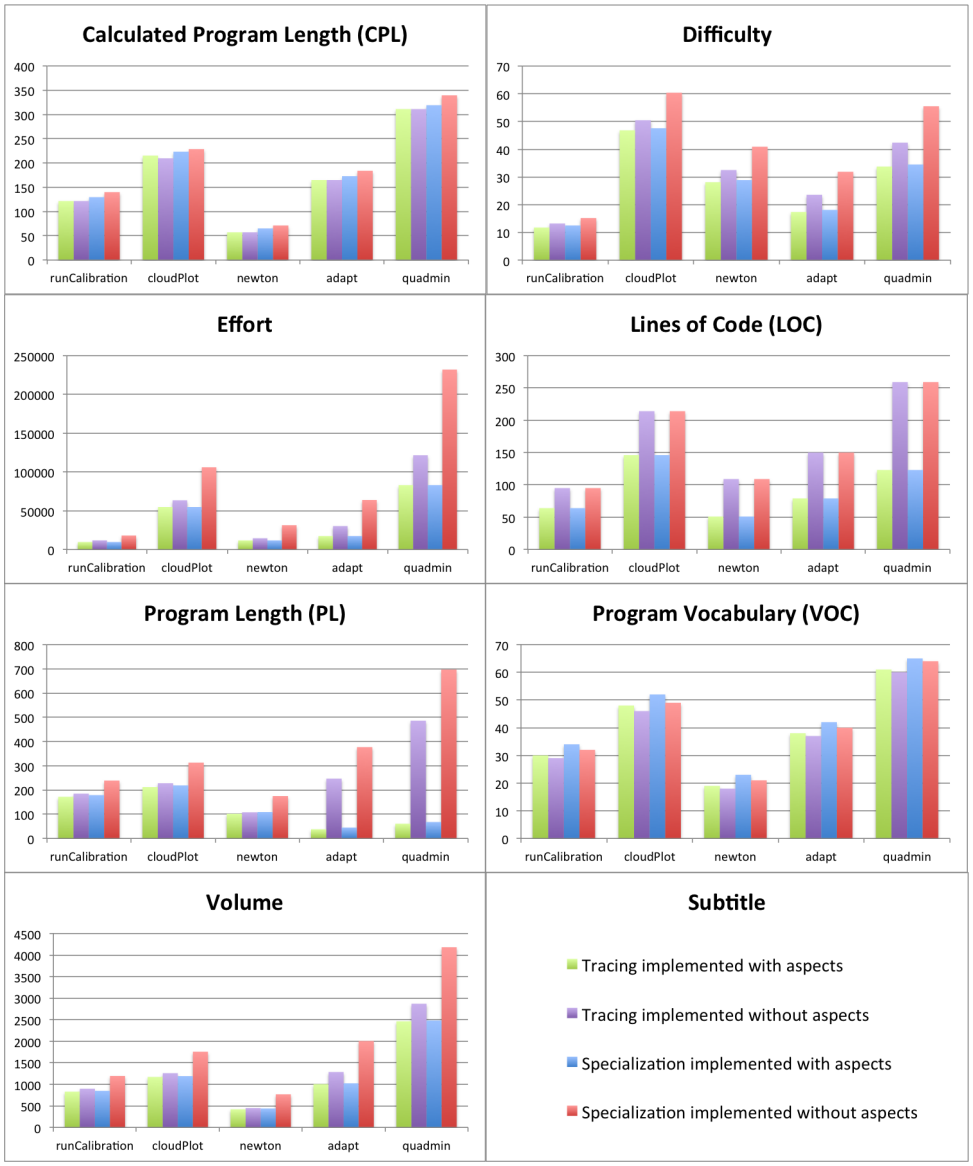

Figure 5: Metrics results for each function with tracing and specialization, implemented with and without aspects.

We followed the same analysis strategy we showed throughout Section 5, first analyzing the potential for aspects implementation, and secondly analyzing the changes on the final solutions improved with tracing and specialization when

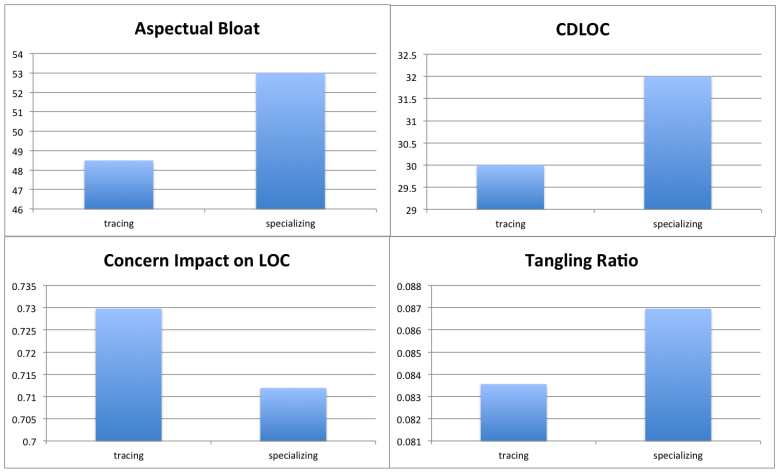

Figure 6: Aspect metrics in IMPACTED.

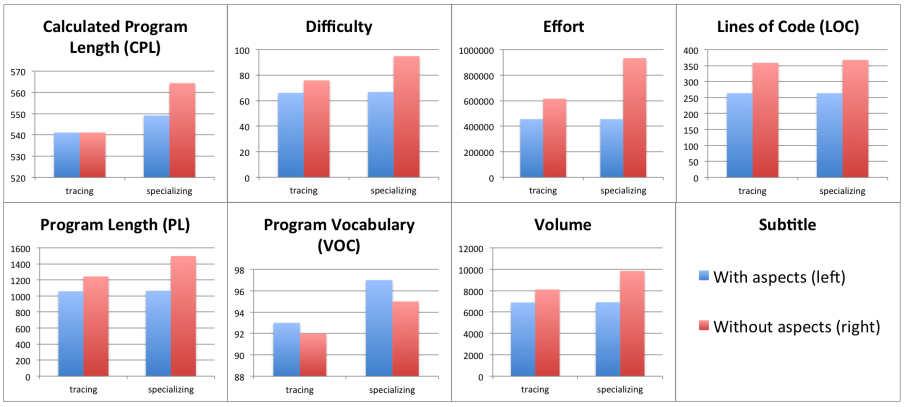

Figure 7: Metrics for IMPACTED with tracing and specialization, implemented with and without our aspects language.

aspects were added to the programming cycle. Figure 6 and Figure 7 show the results of these two analysis, respectively.

The results show an overall improvement in code quality, increasing as much as 21% in LOC and PL. The effort to understand the code shows improvements of 26% and 52% for the versions of IMPACTED where tracing and specializing were introduced. On other charts we see worse resuls, such as the CPL that shows no improvements at all in the version with tracing and only 37% for the specialized version. A metric in particular, the *Vocabulary*, shows an increase in the implementing effort. This might be due the fact that, by introducing a new language, we are forcing the programmer to learn a new set of constructors and

primitives. We do not see this as a pitfall though, as it is a one of task and the overall improvements in all the other metrics prove it is worth it.

6 Implementation

The metrics presented are implemented in our MATLAB front-end. This font-end includes parsers and construction of the abstract syntax tree for MATLAB (with the MATLAB to Tom-IR tool [12]) and for the MATLAB aspect language.

This front-end was developed using advanced language engineering techniques, like generalised (top-down) parsing (using the ANTLR parser generator [17]), strategic programming [18–20] (implemented with the TOM system [21]), attribute grammars [22, 23] (implemented in the Lrc system [24]), and formal program calculation techniques to reason about our implementions [25, 26]. By using these we can easily define tree-traversal algorithms, that we heavily use to weave the abstract data-types of both the aspects and the MATLAB code.

These techniques allowed us to implement all metrics presented in this paper in a concise and generic way. That is, they are independent of the abstract tree. As a consequence, new metrics can be easily added to our metric suite.

7 Conclusion

In this paper we presented software metrics for assessing the software complexity of both standard MATLAB programs and aspect oriented MATLAB programs.

We adapted a set of AOP metrics to the Aspect MATLAB realm, by implementing them in our MATLAB front-end, and used them to assess the complexity of MATLAB programs when compared to their AOP equivalents. We did so by applying our suite first to a set of widely used MATLAB functions and later to a fully developed MATLAB program, consisting of many MATLAB functions.

Our preliminary results are promising and show that aspect oriented programming in MATLAB improves the quality of programs. Both the MATLAB metrics and the aspect metrics are implemented in our (Aspect) MATLAB frontend.

References

1. MATLAB: version 7.10.0 (R2010a). The MathWorks Inc., Natick, Massachusetts (2010)
2. MathWorks: Front page. `http://www.mathworks.com` [Accessed in February 2012].
3. Halstead, M.H.: Elements of Software Science (Operating and programming systems series). Elsevier Science Inc., New York, NY, USA (1977)
4. Aslam, T., Doherty, J., Dubrau, A., Hendren, L.: Aspectmatlab: an aspect-oriented scientific programming language. In: Proceedings of the 9th International Conference on Aspect-Oriented Software Development (AOSD), New York, NY, USA, ACM (2010) 181–192
5. MathWorks: R2012a documentation - fixed-point toolbox. `http://www.mathworks.com/help/toolbox/fixedpoint/ref/quantizer.html` [Accessed in February 2012].
6. Cardoso, J., Fernandes, J., Monteiro, M.: Adding aspect-oriented features to matlab. In: workshop on Software Engineering Properties of Languages and Aspect Technologies (SPLAT! 2006). (2006)

7. Cardoso, J., Diniz, P., Monteiro, M.P., Fernandes, J.M., Saraiva, J.: A domain-specific aspect language for transforming MATLAB programs. In: Fifth Workshop on Domain-Specific Aspect Languages (DSAL). (March 2010)
8. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of aspectj. In: Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP), London, UK, Springer-Verlag (2001) 327–353
9. Peckhan, J., Lloyd, S.J.: Practicing Software Engineering in 21st century. IRM Press (2003)
10. Lopes, C.V.: D: A Language Framework for Distributed Programming. PhD thesis, College of Computer Science, Northeastern University (1997)
11. Sant'anna, C., Garcia, A., Chavez, C., Lucena, C., v. von Staa, A.: On the reuse and maintenance of aspect-oriented software: An assessment framework. In: Proceedings XVII Brazilian Symposium on Software Engineering (SBES). (2003)
12. Nobre, R., Cardoso, J.M.P., Diniz, P.C.: Leveraging type knowledge for efficient matlab to c translation. In: 15th Workshop on Compilers for Parallel Computing (CPC). (2010)
13. MathWorks: Matlab central - file exchange. `http://www.mathworks.com/matlabcentral/fileexchange` [Accessed in February 2012].
14. Ribeiro, J.D.S.R.G.J.R., Pais, T.C.: Hazard avoidance developments for planetary exploration. 7th International ESA Conference on Guidance, Navigation and Control Systems (2008)
15. Reynaud, S., Drieux, M., Bourdarias, C., Philippe, C., Pham, B.v., Transportation, A.S.: Science driven autonomous navigation for safe planetary pin-point landing 1. Context (2009) 1–10
16. Pais, T., Ribeiro, R.A.: Contributions to dynamic multicriteria decision making models. Proceedings of the International Fuzzy Systems Association World Congress and European Society for Fuzzy logic and technology Conference (IFSA-EUSFLAT) (2009) : 719–724
17. Parr, T.: The Definitive ANTLR Reference: Building Domain-Specific Languages. First edn. Pragmatic Programmers. Pragmatic Bookshelf (2007)
18. Visser, J., Saraiva, J.: Tutorial on strategic programming across programming paradigms. In: 8th Brazilian Symposium on Programming Languages (SBLP). (2004)
19. Balland, E., Moreau, P.E., Reilles, A.: Rewriting strategies in java. Electron. Notes Theor. Comput. Sci. **219** (2008) 97–111
20. Lämmel, R., Visser, J.: Program transformation with stratego/xt: Rules, strategies, tools, and systems in strategoxt-0.9. In et al., L., ed.: Domain-Speciffic Program Generation. LNCS, Springer-Verlag (2003)
21. Balland, E., Brauner, P., Kopetz, R., Moreau, P.E., Reilles, A.: Tom: Piggybacking rewriting on java. In: Term Rewriting and Applications. LNCS, Springer-Verlag (2007)
22. Knuth, D.E.: Semantics of Context-free Languages. Mathematical Systems Theory **2**(2) (1968) 127–145 Correction: *Mathematical Systems Theory* 5, 1, pp. 95-96 (March 1971).
23. Saraiva, J., Swierstra, D.: Generating Spreadsheet-like Tools from Strong Attribute Grammars. In Pfenning, F., Smaradakis, Y., eds.: ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE). Volume 2830 of LNCS., Springer-Verlag (2003) 307–323
24. Kuiper, M., Saraiva, J.: Lrc - A Generator for Incremental Language-Oriented Tools. In Koskimies, K., ed.: 7th International Conference on Compiler Construction (CC/ETAPS). Volume 1383 of LNCS., Springer-Verlag (1998) 298–301
25. Fernandes, J.P., Pardo, A., Saraiva, J.: A shortcut fusion rule for circular program calculation. In: ACM SIGPLAN Haskell Workshop. Haskell'07, New York, NY, USA, ACM (2007) 95–106
26. Pardo, A., Fernandes, J.P., Saraiva, J.: Shortcut fusion rules for the derivation of circular and higher-order programs. Higher-Order and Symbolic Computation (2011) Springer, 1–35