Identification and Characterization of Crosscutting Concerns in MATLAB Systems[1]

Miguel P. Monteiro[1], João M. P. Cardoso[2], Simona Posea[3]

[1, 3] CITI, Departamento de Informática, Faculdade de Ciências e Tecnologia Universidade Nova de Lisboa, 2829-516 Caparica, Portugal
[1]mmonteiro@di.fct.unl.pt, [3]simona8810@yahoo.com
[2] Departamento de Engenharia Informática, Faculdade de Engenharia (FEUP), Universidade do Porto, 4200-465 Porto, Portugal, [2]jmpc@acm.org

Abstract. The current state-of-the-art in aspect mining is well advanced for object-oriented programming languages but until now neglected the MATLAB language. This paper contributes to fill that gap by proposing a novel notion of crosscutting concern, tailored for the specific characteristics of MATLAB code bases. We present an exploratory, token-based, approach to aspect mining for MATLAB. An analysis of data obtained from a tool using this approach over 209 publicly available MATLAB files indicate the approach is valid for detecting several kinds of crosscutting concerns in MATLAB systems.

Keywords: MATLAB, aspect mining, code tangling, crosscutting concerns.

1 Introduction

Currently, the state-of-the-art of *aspect mining* [6], i.e., identifying and locating crosscutting concerns (CCCs) as latent aspects in source code, is well advanced for object-oriented (OO) programming languages such as Java. However, most research on aspect mining has until now neglected the popular and widely used procedural language MATLAB. We contribute to address this gap by presenting an exploratory effort to identify and characterize CCCs in MATLAB systems. This paper argues that, due to the specific characteristics and different typical uses of MATLAB, a rethink of the notion of crosscutting concern is warranted, which also requires fresh approaches for their identification in MATLAB code bases.

This paper highlights some differences between symptoms of the presence of CCCs in OO source code and MATLAB code and proposes a simple but effective, token-based, approach for identifying and locating CCCs in MATLAB code. We present an exploratory study of CCCs in a number of real MATLAB applications in the domain of signal and image processing, as well as a short analysis of data collected from that study.

[1] Work for the present publication was partially supported by project AMADEUS under grant (POCTI, PTDC/EIA/70271/2006) from FCT (Portuguese Science Foundation).

The rest of this paper is organized as follows. Section 2 reviews traditional notions of CCC. Section 3 outlines our approach to tackle CCCs in the specific case of MATLAB and on that basis proposes a novel notion of CCC, tailored for MATLAB. Section 4 analyses the suitability of current of aspect mining techniques to be used as an initial, exploratory approach for aspect mining for MATLAB systems. Section 4 proposes an adapted version of one of those techniques for that purpose. Section 5 analyses data collected from MATLAB systems in the domain of signal and image processing. Section 6 outlines opportunities for future work and concludes the paper.

2 Crosscutting concerns and aspect-oriented programming

It has long been accepted that existing programming paradigms suffer from the *tyranny of the dominant decomposition* [12], meaning that each programming paradigm provides a *single* decomposition criterion for a software system. As a consequence, concerns that do not align with the primary decomposition tend to cut across the decomposition units. Such non-aligning concerns are known as *crosscutting concerns* [8]. The usual symptoms of the presence of a CCC in source code are *code scattering* and *code tangling* [8].

Aspect-oriented programming [8] was proposed as an approach to modularize CCCs and thus eliminate the negative symptoms of scattering and tangling. The majority of aspect-oriented languages are backwards-compatible extensions of existing languages, with a marked predominance of OO languages such as Java. Typically, such language extensions add a distinct kind of (often class-like) module – the *aspect* – for the specific purpose of enclosing code related to CCCs.

In OO systems, decomposition units are classes as full-fledged modules. In such systems, code scattering usually takes the form of code fragments scattered across multiple units of modularity, often corresponding to repeated instances of “boiler plate code”. Tangling is found in the modules that the CCCs overlap: code pertaining to the primary concern appears intertwined with code pertaining to other concerns. Tangling is particularly harmful to the comprehensibility of all concerns found in the unit, including the primary concern. A Java example is given in section 3.

In less structured programming paradigms such as that supported by (mainly procedural) MATLAB, decomposition units are simpler, less powerful constructs. Since the MATLAB language features are very different from those available in typical OO languages, it is reasonable to expect that symptoms of CCCs and indeed the very idea of CCC requires some adaptation to the different context.

In principle, any approach to identify and locate CCCs is directly dependent of whatever notions are held of what is a CCC. Such notions are traditionally based on the capabilities of the specific aspect-oriented extension of the language concerned. However, that approach assumes an already existing, clearly defined AOP extension to the language. The present situation differs somewhat for two reasons: (1) the distinct characteristics of MATLAB (see section 3), and (2) the fact that in this case, development of the infrastructure to support the language extension is ongoing [4]. The design of our infrastructure enables a wide range of choices in its future development, which can be geared to tackle whatever turns out to make a promising aspect. Our primary criterion for selection is *impact*: the uses that promise to be useful to most MATLAB users comprise the most desirable targets for future development.

The work presented in this paper is part of an ongoing project whose primary aim is to create an aspect-oriented extension to the MATLAB programming language [5]. The approach taken is focused on the support to separate *aspect modules* that specify functionality that would otherwise cut across, or "pollute", the core parts of the system (MATLAB functions) giving rise to the tangling/pollution symptom.

A detailed description of the infrastructure and underlying approach is not required to understand our proposed notion of CCC. For such an description we refer to our DSAL 2010 paper [4]. It suffices to know the following:

- A *transformational approach* is taken, in which *aspect weaving*, i.e., composition of the aspect modules to the core parts of the application, is carried out by transformation tools according to rules specified in the aspect modules. The outcome of the aspect weaving is transformed, legal MATLAB code.
- The approach is more invasive than traditional aspect-oriented approaches such as AspectJ [7]. Virtually any detail in the base code is potentially the target of some transformation. The transformed code can be considerably different from the code from the base, original, system.
- Potential uses for the transformations supported cover a wide range of concerns, from monitoring, profiling, debugging, to the configuration of variables to support non-standard *shapes*, i.e., numerical representations.
- The infrastructure was designed to support the addition of new, *composition rules* that facilitate a routine development of *case-specific*, *throw-away aspects*.

This approach markedly differs from the usual, compiled approaches to AOP. It also differs from AspectMatlab [1], an extension of MATLAB specifically developed for scientific programming. Our infrastructure targets a significantly wider range of domains. Its transformational approach is motivated by the specific characteristics of MATLAB, particularly its interpreted nature, which blurs the usual distinction between *source* code and *executable* code. MATLAB code plays both roles: as a parallel with Java, the MATLAB code generated by the infrastructure can be regarded as a product of *source-code transformation*, but it is equally reasonable to approach it as the outcome of *bytecode weaving* [7].

3 A notion of crosscutting concerns in MATLAB

In addition to its interpreted nature, the characteristics of MATLAB differ from languages typically extended to aspect-orientation in other respects that also have an influence on the specific details of the symptoms of the presence of CCCs. The procedural nature of MATLAB means that its decomposition units are predominantly individual functions and groups of functions called toolboxes. MATLAB's OO extensions comprise a recent add-on: typical MATLAB applications do not use them. Since our primary criteria for the selection of problems to be tackled is impact, issues pertaining to MATLAB's OO features are not promising candidates. It is worth noting that OO features are also ineffective in modularizing the concerns that are the focus of this paper, often for the same reasons why they also appear in OO systems.

Simple functions and groups of functions comprise a less powerful, more limited mechanism for modularizing concerns than class modules, so it is to be expected that concerns will not appear so well organized within MATLAB systems. Typical uses of MATLAB also differ from those of typical OO languages, so they may give rise to different code symptoms. Therefore, studies of the actual code symptoms in existing MATLAB systems are warranted, namely to characterize the precise ways in which they differ from symptoms in OO code, as well as the typical underlying causes. To our knowledge, no such studies have been carried out until now. The sole prior description of symptoms of the presence of CCCs in MATLAB code is provided by Cardoso et al [5]. Their example comprises the configuration of a benchmark to a specific fixed-point representation. The description provided includes snippets of a clean, version of a function and a "polluted" version using a specific fixed-point representation, programmatically supported. The example suggests fine-grained shape configuration may be a typical CCC in MATLAB.

To illustrate the proposed notion of CCCs in MATLAB and highlight differences between symptoms of CCCs in OO languages and symptoms of CCCs in MATLAB systems, we next describe a simple example of CCC in each platform. In both cases, the secondary concern is the graphical presentation of data, i.e., a *display concern*. However, the specific details for each case are markedly different, reflecting the differences between the Java and MATLAB platforms, as well as their typical uses.

3.1 A crosscutting concern in Java

Fig. 1 shows the classes comprising an example often used to illustrate the effects of CCCs in Java systems, including in the seminal paper on AspectJ [7]. It comprises an abstract declaration of a `FigureElement` type, plus a number of classes – `Point` and `Line` – that concretize it. The classes provide operations for changing the values of the figure elements. Also included is a display functionality to provide a graphical presentation of the figure elements that must be updated upon all changes to them. This is represented by class `Display`, also shown in Fig. 1.

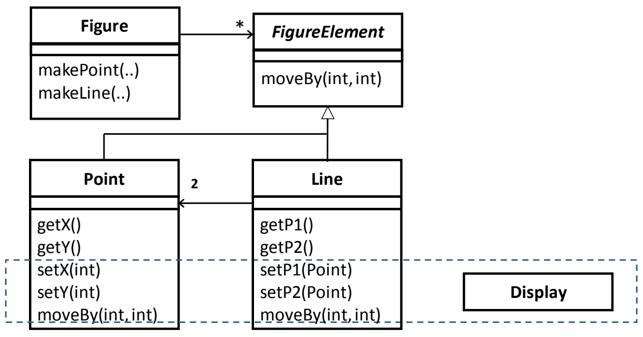

Fig. 1. The Figure example used in [7]

The problem is felt when trying to compose the update logic to the existing decomposition units (here: classes). All operations that have an impact on the graphical representation of the figures must undergo invasive changes, in the form of the references to `Display`. Fig. 2 shows the effect on source code of class `Point`, with code pertaining to the secondary concern highlighted in grey background. We see code related to the display concern in addition to code related to the module's core concern. Similar symptoms would be found in other `FigureElement` classes (not shown). Note that in real cases, multiple CCCs are often found in single modules.

```java
public class Point implements FigureElement {
  private int _x, _y;
  private Display _display;

  public Point(int x, int y) {
     _x = x;
     _y = y;
  }
  public Point(int x, int y, Display display) {
     this(x, y);
     setDisplay(display);
  }
  public void setX(int x) {
     _x = x;
     _display.update(this);
  }
  public void setY(int y) {
     _y = y;
     _display.update(this);
  }
  public void setDisplay(Display display) {
     _display = display;
  }
  public void moveBy(int dx, int dy) {
     _x += dx;
     _y += dy;
     _display.update(this);
  }
}
```

Fig. 2. Illustrative example of a CCC in Java

3.2 A crosscutting concern in MATLAB

Fig. 3 shows a MATLAB function in two versions. The value returned for a given parameter value is the same in both cases. At the left a clean version is shown, whose code relates to its core concern exclusively – compute the result of the exponential function applied to a specific parameter value using the first N terms of the power series expansion. At the right, a tangled version is shown also including a display concern, MATLAB style: preparing a call to function 'plot'. The code pertaining to the secondary concern is highlighted in grey, as in Fig. 2. The extra code is mostly about building the vector data required for building a two-dimensional representation of the function within a given range, which will feed 'plot' at the end. The tangled version defines an additional (and of optional use) parameter to support a choice between creating and not creating the plot representation. The sole motivation for this parameter is support to one of the primary advantages of modularity: (un)pluggability of the functionality concerned. Note that in real-world examples, use of ‘plot’ would pollute the original code even more, as programmers usually also include a title and legends in the plotted figure.

This example differs from the Java example in one crucial point: the secondary concern does not merely comprise additional code, intertwined with the original code (i.e., tangling). Rather, additional functionality has a direct impact on the original code that yields *different, modified code*. The computation is no longer performed on simple, scalar values. Instead, vectors are used in place of the original variables to produce the data that will feed ‘plot’.

Recent versions of MATLAB incorporate object-oriented features. However, it is not possible to reuse the original version from this example, even through inheritance.

This MATLAB example is of course a trivial one, having been taken from tutorial material used for teaching programming at the institution to which one author is affiliated. However, the course is not controlled by the authors and the example is presented as is. That such a clear example of a CCC can be found in tutorial material serves to highlight how pervasive this kind of symptoms is in MATLAB systems.

```
function z = expo(x,n)
  y = 1;
  for i = 1:n
    y = y + x^i/factorial(i);
  end
  z = y;
```

```
function z = expo(x,n,p)
  P(1) = 1;  Y(1) = 1;
  for i = 1:n
    P(i+1) = P(i)+1;
    Y(i+1) = Y(i) + x^i/factorial(i);
  end
  z = Y(n+1);
  if (p) plot(P,Y) end
```

Fig. 3. Illustrative example of a CCC in MATLAB

3.3 A notion of crosscutting concerns in MATLAB

This paper builds upon the work by Cardoso et al [5] and proposes a notion of a CCC in MATLAB that covers any case in which a given decomposition unit – for brevity, henceforth just “function” – encloses code that can conceptually be traced to more than one concern. The stress on the conceptual (semantic) level is important: a CCC in a function is always a *secondary* concern that is found in addition to the function’s *core concern*. In all such cases, the secondary concern can conceivably be extracted from the function, yielding a “clean” version of the function in which only code pertaining to the core concern remains. However, in MATLAB systems it must be assumed that the presence of a secondary concern yields *different* code from the original, which also directly depends on the specifics of the secondary concern.

It is important to note that a significant part of the secondary concerns that fall into the proposed definition can conceivably be implemented by means of some additional language feature – such as can be supported by our infrastructure. Thus, a promising indicator of a potential aspect is to feel the need or desirability of some unsupported additional feature, of a possibly narrow applicability. Though narrowly case-specific, modularization and (un)pluggability of such features may bring significant benefits to developers.

Discovering some of these concerns can be a subtle task. For instance, Fig. 4 shows a fragment from one of the systems we analyzed. ('drawedgelist'). It is used to assist debugging and requires an assignment to variable 'debug', defined in the MATLAB code of the function with a default value of 0 (i.e., `false`).

We identified the following preliminary list of CCC categories upon manual inspection of a number of real cases, complemented with insights acquired upon analysis of data collected from a tool (presented in the next section):

- *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.

```
if debug
  for I = 1:Nedge
      mid = fix(length( edgelist{I} ) / 2);
      text( edgelist{I}(mid,2), edgelist{I}(mid,1), sprintf('%d',I) )
...
```

Fig. 4. Segment of MATLAB code for debugging purposes.

In the example presented in Fig. 5 (a fragment of function 'gaborconvolve'), we show a number of aspect candidates whose behavior deals with argument verification ('nargin'), class verification ('isa'), freeing memory ('clear'), and messages to the user ('fprintf'). In this case, code related to debugging can be difficult to automatically expose as an argument variable ('feedback') is used as an option to enable the printing of certain messages in the screen. An advanced aspect mining tool should identify the use of functions to print messages to the screen ('fprintf' in this case) and the code associated with those functions.

```
function EO = gaborconvolve(im, nscale, norient, minWaveLength, mult,
... sigmaOnf, dThetaOnSigma, feedback)
  if nargin == 7
    feedback = 0;
  end
  … % original code removed
  if ~isa(im,'double')
    im = double(im);
  end
  … % original code removed
  clear x; clear y; clear theta; % save a little memory
  … % original code removed
  for o = 1:norient, % For each orientation.
    if feedback
       fprintf('Processing orientation %d \r', o);
    end
  … % original code removed
  end
  if feedback, fprintf('      \r'); end
```

Fig. 5. Illustrative example of CCCs in MATLAB

4 Aspect mining

The task of identifying and locating CCCs in existing code is called *aspect mining* [6]. Typical research on aspect mining covers the development of methods and tools for the automatic or semi-automatic detection and identification of concerns that comprise promising candidates for being extracted to their own aspect modules. Once identification is concluded, a refactoring process can be performed for the extraction of aspects yielding an aspect-oriented version of the original target system [10].

To date, research on aspect mining is largely focused on the OO paradigm, plus the specific case of the C programming language [3]. Java is the most usual OO representative [6]. We have no knowledge of any previous approach on aspect mining for MATLAB systems. Moreover, it turns out that MATLAB code places different, possibly more challenging hurdles for aspect mining tasks.

This section next briefly surveys existing approaches to aspect mining [3], [9], [2], [11]. The intent is not to issue a final judgment on those techniques, even in relation to MATLAB. We aim to select the approach that promises to be the most suitable for an *initial*, *exploratory* study in a topic in which many facets remain unclear.

Two techniques focus on the analysis of method calls, respectively static and dynamic: fan-in analysis [9] and analysis of event traces [2]. Both work at an abstraction level and granularity suitable for object-oriented systems. However, MATLAB does not support the equivalent of full Application Program Interfaces (APIs) in OO systems. Most MATLAB elements available for analysis are minute details of function implementations that are hidden from APIs, such as names of local variables, control structures and names of called functions. Indeed, it is not clear that either technique would be effective for cases like the one described is section 3.

Another technique is the extraction of conceptual knowledge from names in code [11]. Unfortunately, the typical style of naming prevalent in the MATLAB community seems an impediment to that kind of approach. Developer support in MATLAB tools is less rich than that provided for the Java community and lacks features such as code completion and refactoring. Due to such circumstances, programmers tend to use short, cryptic names for parameters and variables, often with a single letter. Function names are longer, but with some exceptions they generally comprise a single word (e.g., 'zeros', 'values') or a cryptic combination of a few shortened words/acronyms (e.g., 'cumtrapz', 'tsearchn'). Thus, high-level, domain concepts are more poorly represented in MATLAB than in Java. For this reason, extraction of conceptual knowledge does not seem promising as an initial approach.

Another option are clone detection techniques [3]. As argued in section 3, the presence of secondary concerns in MATLAB often results in modified code whose details depend on both the original primary logic and the secondary concern in question. This results in higher levels of variability of symptoms in the code, as compared to what is observed in OO systems. One can imagine multiple instances of one same function, each differently "transformed" by a different secondary concern. There is no guarantee that each such instance will bear enough similarities for approaches based on resemblances of code fragments. Thus, clone detection techniques do not generally look suitable for detecting and identifying CCCs in MATLAB code, with one exception.

4.1 Aspect mining tailored to MATLAB

Token-based clone detection techniques are about performing a tokenization of the source code and subsequently use the tokens as a basis for clone detection [3]. Our approach is an adapted version of this kind of clone detection technique, which nevertheless allows us to deal efficiently with all MATLAB examples, regardless of language version. Target systems are assumed to be MATLAB-grammar compliant and no checks for syntactic or grammatical correctness are performed.

The core idea is to count occurrences of function calls in non-comment text lines. The tool decomposes MATLAB source files into sequences of tokens and computes metrics based on those tokens. Distinguishing variable accesses from function calls is not straightforward in MATLAB, as the same syntax is used for both. Thus, the tool builds a list of variable names by extracting them from the function declarations, and by analyzing each individual line to determine if it is a variable assignment. If it is, it registers the *lvalue*, i.e., the name in the left side of the assignment. The list of variable names thus collected is used to filter out the collection of names extracted from the source text. It is assumed that the remaining names are function names. For each separate MATLAB file, a number of metrics is computed, including:

1. Number of times a given function name appears in a given MATLAB file.
2. Number of different function names appearing in a given MATLAB file.

The tool can compute the above metrics for individual `*.m` files, all `*.m` files in a toolbox and all `*.m` files in arbitrary collections of toolboxes. Further functionality was included to ensure that computed data is presented in a form that scales to a fairly large number of systems. Output comprises several tables with this data.

5 Analysis of collected data

Though language processing performed by the tool is not sophisticated, the metrics collected proved very useful. Note that a low number of occurrences cannot count as an useful indicator, as it is perfectly reasonable for a function to call another function several times. However, it became clear that a "high" number of calls is a fairly reliable indicator of tangling. For instance, this metric yields value 8 for the example presented by Cardoso et al [5] (not shown due to space constraints). Determining more precise thresholds is an obvious next step that is left for future work.

We computed the aforementioned metrics for a collection of MATLAB repositories (i.e., applications) in the signal and image processing domains, spanning 17 repositories with 209 MATLAB files and a total of 7,775 lines of code. Fig. 6 shows the number of uses of candidate functions and percentage of the most representative candidates. Function 'size' is the most often used: 194 uses that correspond to 15% of the global uses of the functions identifying aspect candidates. Functions 'error', 'zeros' and 'nargin' are also heavily used (*147×, 11.4%*, *123×, 9.5%* and *89×, 6.9%* respectively). Note that a more advanced analysis would be required to pinpoint the uses of 'zeros' used exclusively to allocate memory, as there are cases in which the function is used to actually initialize a matrix with zero values.

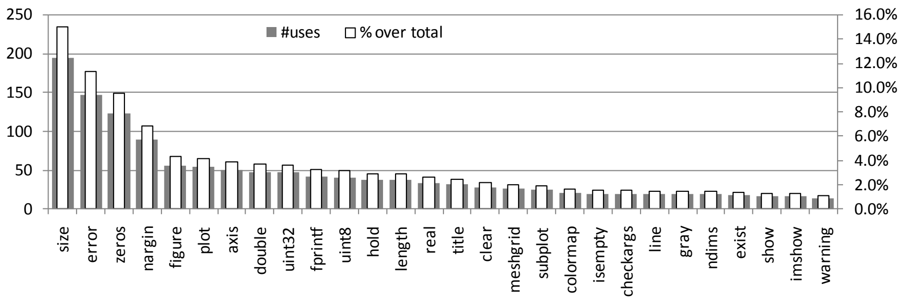

Fig. 6. Aspect candidates present in the analyzed repositories.

The MATLAB files analyzed include the use of 28 functions that we selected as candidates to aspects. Those functions appear 1294 times in the 209 files analyzed, yielding an average of 6.19 uses per `*.m` file. If we measure the "pollution" level as the number of uses per lines of code we have for these examples, around 16.64% of lines use functions that we cataloged as related to aspects. Considering scattering results (i.e., simply considering if a function is used or not in a MATLAB file), there are a total of 646 references which yields about 3.09 different functions referred per file. Fig. 7 shows the frequency of `*.m` files per range of candidate functions. Of 209 files, just 19 do not use candidate functions and 116 files use from 1 to 4 candidate functions. These results indicate a significant proportion of MATLAB code that comprises promising candidates for extraction to future aspectual extensions of MATLAB.

It is worth noting that the MATLAB systems analyzed reveal a low use of shape specialization. This is natural, as most publically available MATLAB systems use high-level models and few of these systems include specialized versions, needed for any case specific use of those systems, e.g., in embedded systems. Most available code is for simulating and problem-solving under the MATLAB environment and is not publicly provided to become part of final system implementations.

Some functions include additional code to make them as generic as possible. When specific specializations are needed, the extra code needed to generalize those functions can be made (un)pluggable, yielding a cleaner version of the core function.

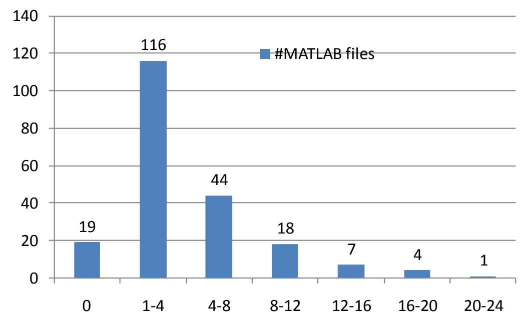

Fig. 7. Frequency of MATLAB files per functions candidate to aspects.

MATLAB models also usually use input/output features during development that will not be used in the final, working system. These include reading from and writing to files, which should be removed and replaced by a suitable input/output aspect.

In some cases, functions are part of a specific behavior and a more elaborate analysis would have to be performed to indentify the code segments that relate to that behavior. For instance, 'nargin' is often used to deal with optional arguments. In the repositories used for this study, a function named 'phasecong' includes 32 lines related to 'nargin' verification and initializations according to the number of arguments passed in the function call. This kind of behavior can conceivably be extracted to a "specialization aspect". That code could then be removed from the MATLAB source and an invocation of the function with fewer arguments could originate a specialization of that function. Such specialization includes the code that should be executed for a certain number of arguments. Another case is when the *class* (i.e., MATLAB type) of a variable is verified and according to the *class* a specific behavior is invoked.

6 Conclusion and future work

This paper described our first contribution to aspect-mining in MATLAB systems, proposing a novel notion of crosscutting concern for MATLAB and evaluating whether instances of those concerns are found in a significant number of MATLAB files. Experimental results using a token-based aspect-mining tool and targeting publicly available real systems enabled the identification of a number of promising candidates for modularization through extraction to aspect modules.

There are many directions through which this work can be developed in future, contributing to refine and mature the notion of crosscutting concern proposed in this paper. One front is to explore the (semi-)automatic identification of crosscutting concerns ("candidate aspects") on the basis of broader code segments, namely through pattern matching. However, much analysis remains to be performed on data computed through this simple, token-based approach. The "number of calls per individual function" metric proves an useful indicator of tangling, but this hypothesis requires further analysis and assessment. In addition, there are several other hypotheses that can be assessed through this approach, namely: (1) the extent to which "number of functions that call this function" can be a reliable indicator of scattering; (2) whether certain groups of tokens tend to be used together; (3) whether certain tokens tend to appear in connection to specific functionalities; (4) whether specific groups of tokens can be traced to specific application domains.

In a different front, the previously aspect mining techniques surveyed in section 4 should be also explored, to assess in practice the extent to which they can advantageously replace or complement the approach proposed in this paper.

References

1. Aslam T., Doherty J., Dubrau A., Hendren L. AspectMatlab: An Aspect-Oriented Scientific Programming Language. AOSD 2010, Saint Malo, France, March 2010.
2. Breu S., Krinke J. Aspect Mining Using Event Traces. 19th IEEE international conference on Automated software engineering (ASE 2004). Linz, Austria, September 2004.
3. Bruntink M., Deursen A. van, Engelen R. van, Tourwé T. An Evaluation of Clone Detection Techniques for Identifying Cross-Cutting Concerns. ICSM'04, Chicago, Illinois, USA, September 2004.
4. Cardoso, J., Diniz, P., Monteiro, M., Fernandes, J., Saraiva, J. A Domain-Specific Aspect Language for Transforming MATLAB Programs. DSAL 2010, Rennes, March 2010.
5. Cardoso, J., Fernandes, J., Monteiro, M. 2006. Adding Aspect-Oriented Features to MATLAB. SPLAT!2006, Bonn, Germany, March 2006.
6. Kellens A., Mens K., Tonella P. A Survey of Automated Code-Level Aspect Mining Techniques. Transactions on Aspect Oriented Software Development, vol. 4 (LNCS 4640), pp. 145-164. Springer, 2007.
7. Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., Griswold W., An Overview of AspectJ. ECOOP 2001, Budapest, Hungary, June 2001.
8. Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C., Loingtier J., Irwin J., Aspect-Oriented Programming. ECOOP'97, Jyväskylä, Finland, June 1997.
9. Marin M. Deursen A., Moonen L., Identifying Aspects using Fan-In Analysis. Working Conference on Reverse Engineering (WCRE2004), Delft, The Netherlands, Nov 2004.
10. Monteiro M.P., Fernandes J.M. Towards a Catalogue of Refactorings and Code Smells for AspectJ. LNCS TAOSD I, Springer vol. 3880, 2006.
11. Shepherd D., Fry Z., Hill E., Pollock L., Vijay-Shanker K. Using Natural Language Program Analysis to Locate and Understand Action-Oriented Concerns. AOSD 2007, Vancouver Canada, March 2007.
12. Tarr P., Ossher H., Harrison W., Sutton Jr., S.M., N Degrees of Separation: Multi-Dimensional Separation of Concerns. ICSE'99, Los Angeles, USA, May 1999.