

A Multiple View Interactive Environment to Support MATLAB and GNU/Octave Program Comprehension

Ivan de M. Lessa, Glauco de F. Carneiro
Salvador University (UNIFACS)
Salvador/Bahia, Brazil
ivan.lessa@gmail.com,
gluaco.carneiro@unifacs.br

Miguel Jorge T. P. Monteiro
New University of Lisbon (UNL)
Lisbon, Portugal
mtpm@fct.unl.pt

Fernando Brito e Abreu
Lisbon University Institute (ISCTE-IUL)
Lisbon, Portugal
fba@iscte-iul.pt

Abstract— Program comprehension plays an important role in Software Engineering. In fact, many of the software lifecycle activities depend on program comprehension. Despite the importance of MATLAB and Octave programming languages in the Engineering and Statistical communities, little attention has been paid to the conception, implementation and characterization of tools and techniques for the comprehension of programs written in these languages. Considering this scenario, this paper presents a Multiple View Interactive Environment (MVIE) called *OctMiner* that supports the comprehension of programs developed in the aforementioned languages. *OctMiner* provides a set of coordinated visual metaphors that can be adjusted in accordance with the comprehension goals. An example is presented to illustrate the main functionalities of *OctMiner* in a real scenario of program comprehension.

Keywords- Program Comprehension; Software Visualization; MATLAB; Octave; Crosscutting Concerns.

I. INTRODUCTION

Program comprehension is about understanding how a software system or a part of it works [15]. The strategies followed to understand software might vary among developers depending on their personality, experience, skills, task at hand, or technology used [15]. The literature has pointed out the need to provide support to the comprehension of MATLAB and Octave programs [5][12]. MATLAB (acronym for Matrix Laboratory)¹ is an interpreted and imperative programming language with focus on matrix data types and operations on them. Octave² is the General Public License version of the MATLAB programming language. MATLAB and Octave are quite similar; hence programs written in both languages are easily portable to one another. These two languages are used in scientific computing, control systems, signal processing, image processing, system engineering, simulation, among other fields [2][5][9]. MATLAB and Octave programs comprise 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, but those that are used are passed by value. Scripts correspond to files with code without specifying inputs and outputs. Scripts can be also called by other scripts and functions [2][9].

Multiple view interactive environments (MVIEs) provide visualization resources that can support programmers to analyze and understand source code. The views available in the MVIE can be configured in real time to better fit the comprehension needs [4]. Considering the motivation presented in this paper, we conceived and tailored a MVIE called *OctMiner* to visually represent MATLAB and Octave routines. The goal is to use visualization resources to support their comprehension using multiple views. The main reason for this is that a single visual metaphor may not be sufficient to portray the relevant peculiarities of many of the routine properties [4]. A visual metaphor is a paradigm used to visually represent a scene, a situation, or an entity through a structured shape and organized as a map, a tree, a graph, among others [3].

This paper is structured as follows: section II describes the key functionalities of MVIEs to support software comprehension activities; section III summarizes the main concepts of the MATLAB and Octave languages; section IV presents *OctMiner* main architecture structure; section V discusses crosscutting concerns (CCCs) in the context of MATLAB and Octave programs; section VI summarizes the rationale used to map real to visual attributes in *OctMiner*; section VII describes an example of use of *OctMiner* to support the understanding of MATLAB and Octave programs. Finally, section VIII presents the final considerations and future work.

II. MULTIPLE VIEW INTERACTIVE ENVIRONMENTS

Visualization is a means of providing perceivable cues to several aspects of the data under analysis to reveal patterns and behaviors that would otherwise remain hidden [11]. Card et al. [1] proposed a well-known reference model for information visualization. According to them, the creation of views goes through a sequence of successive steps: pre-processing and data transformations, visual mapping and view creation. Carneiro and Mendonça [3] extended this model to adapt it to the context of MVIEs. The extended model is portrayed in Figure 1³ emphasizing that the visualization process is highly interactive. Moreover, it enables the combined use of resources of a multiple view interactive environment. The process starts with original (raw) data obtained from a repository that

¹ <http://www.mathworks.com/products/matlab/>

² <https://www.gnu.org/software/octave/>

³ All the figures of these paper are available at <http://www.sourceminor.org/octminer>

undergoes a set of transformations to be organized into data structures suitable for information exploration. This process is called *data transformation* [3]. Next, the data structures are used to assemble visual data structures. Those structures organize data properties and visual information properties in ways that facilitate the construction of visual metaphors. This step defines the mapping from real attributes – which are derived from the data properties, software attributes, in our case – to visual attributes such as shapes, colors and positions on the screen. This process is called *visual mapping* [3]. It is important to highlight that these activities do not deal with rendering, but rather with building suitable data structures from which the views can be easily computed and rendered. The last step, presented in Figure 1, is the *view transformation*, aimed at drawing the information on the screen to produce the views. In this step, a specific visual scene is actually rendered on the computer screen [3].

Nunes et al. [8] proposed a toolkit implemented as a Java Eclipse plugin from which MVIEs could be developed. The plugin provides a basic structure that allows the creation and inclusion of new resources and functionalities to develop MVIEs. Figure 2 presents the way the toolkit was used and extended by other plugins to comprise the SourceMiner MVIE. This MVIE was originally developed to support the comprehension of Java source code. As can be seen in the figure, the extension points of the *toolkit.aimv* plugin enable the inclusion of new plugins to the MVIE. Each of the extension points conveyed provides an interface with methods and their respective signatures. In the case of *OctMiner*, we needed to access and transform raw data (the Abstract Syntax Tree, well-known as AST, of MATLAB and Octave programs) to a format compatible with the visual data structure. According to the extended reference model for MVIEs, this is a requirement to feed the views.

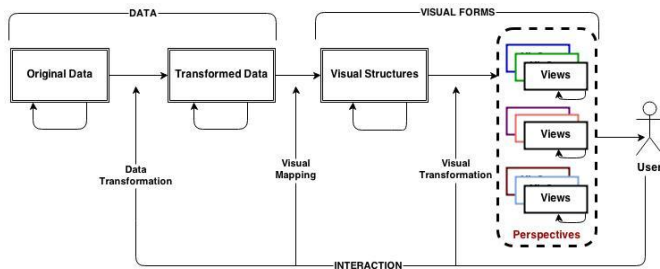


Figure 1. An Extended Reference Model for MVIEs [3]

Figure 2 presents a set of plugins that comprise the SourceMiner MVIE. The following guides are available to help MVIE developers: (1) Data Transformation: to extend the plugin *Import Module* to implement the plugin *sourceminer.modules*; (2) Creating and Applying Filters to extend the plugins *Filter* and *Filter View*; (3) Creating Tools to extend the plugin *Tools*; (4) Creating Views to extend the plugins *Data Views* and *Tools*. These guides are available in the SourceMiner⁴ site. The goal of the toolkit is to provide an infrastructure to develop MVIEs for different domains. The domain targeted in this paper is comprised of programs written in MATLAB and Octave.

⁴ <http://www.sourceminer.org>

III. THE MATLAB AND OCTAVE PROGRAM LANGUAGES

MATLAB is an interpreted language very popular among students and researchers of physics, biomedical engineering and related areas. It is not uncommon that a young engineer is fluent in using MATLAB, but hardly familiar with C, and even less of Fortran [5][12].

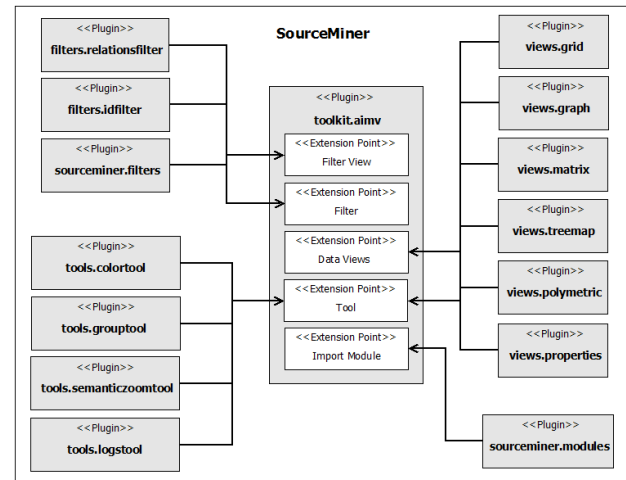


Figure 2. The MVIE SourceMiner [8]

MATLAB has been used to teach linear algebra, numerical analysis, and statistics. Since the MATLAB language is proprietary, a similar language, named Octave was developed, and is distributed under the terms of the GNU General Public License. It was originally conceived in 1988 to be a companion programming language for an undergraduate-level textbook on chemical reactor design. Due to the similarities among the languages, it is possible to interpret MATLAB programs in the interpreter of the GNU/Octave with no major problems. The main differences among the two languages are presented as follows: i) Some similar functions can have different names in each language; ii) Comments in MATLAB are written after “%” while in Octave you can use both “%” and “#”; iii) In MATLAB the control blocks (*while*, *if* and *for*) as well as the functions delimiter all finish with “*end*” while in Octave you can also use “*endwhile*”, “*endif*”, “*endfor*” and “*endfunction*” respectively; iv) In MATLAB the not equal to operator is “*~=*” while in Octave “*!=*” is also valid; v) MATLAB does not accept increment operators such as “*++*” and “*--*”, while Octave accepts them.

IV. OCTMINER: A MVIE FOR MATLAB AND OCTAVE

The main motivation for the visual representation of concerns in a MVIE is the possibility to enhance the comprehension of these programs considering the way these concerns are manifested in MATLAB and Octave programs. For this reason, we present an overview of *OctMiner* architecture in this section. In terms of plugins that interact with the MVIE toolkit, the scenario is the same as presented in Figure 2. Figure 3 depicts the main four elements of *OctMiner*: the Eclipse IDE RAP/RCP, the *Octclipse* plugin, the Octave interpreter and the MVIE toolkit proposed in [8]. Considering that the Eclipse IDE enables its extension through the use of plugins, the MVIE toolkit uses this feature to provide its functionalities as well as to make possible the MVIE tailoring

for the analysis of data from different domains, in this case the data gathered from MATLAB and Octave programs. To achieve this goal we implemented an Analyzer module as conveyed in Figure 3. This analyzer is analogous to the *sourceminer.modules* presented in Figure 2. As can be seen, it is an extension of the Import Module, whose goal is to import and convert data from the original data repository to be represented later in the multiple views. The *Octclipse* plugin converts data from the routine to the Abstract Syntax Tree (AST). From the AST provided by the *Octclipse* plugin we were able to develop a plugin to analyze the AST and to extract data from the routine in a format appropriated to feed the visual structures. This corresponds to the data transformation step of Figure 1. The *Octclipse* plugin also provides an Octave development environment built upon Eclipse's Dynamic Languages Toolkit. This environment enables programmers to create Octave scripts (*.m files), edit them in a multi featured text editor, run the Octave interpreter, and see the result displayed in the IDE's console.

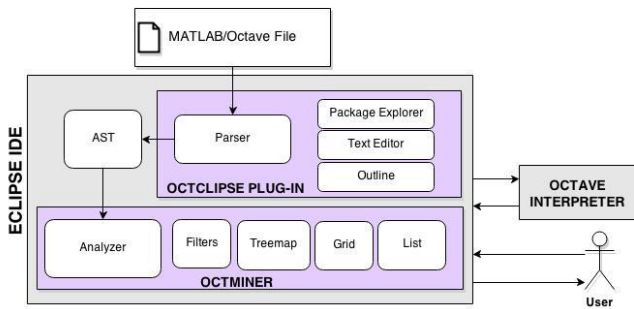


Figure 3: OctMiner Architectural Overview

The AST plays an important role in the context of *OctMiner*. In the AST, all data gathered from the routine is stored in a hierarchy structure. The following statement has its corresponding AST created by the *Octclipse* plugin depicted in Figure 4:

```
x = x-2 * pi * floor (x / (2*pi))
```

As can be seen in Figure 4, the AST provides identifiers for each enclosed element, which are crucial to search the AST. For example, it enables finding out which parameter list a function receives. The AST also provides information about the location of the element in the routine. The Analyzer module gets the data set provided by the AST to feed the visual structures as presented in the visual mapping step in Figure 1. To analyze the structure of statements such as the one presented in Figure 4, we considered the following definitions of each AST element: *ExpressionStatement* (declarations of a given expression); *BinaryExpression* (binary, arithmetic and equality operators); *AssignmentLHSExpression* (classifies the identifiers as variables); *SymbolReference* (usually a variable or function in the sentence); Constant; *SymbolCallExpression* (a function is called through the passage of parameters); *CallArgumentList* (the list of arguments that is passed for the function); *ParenthesisExpression* (indicates a parenthesis in the statement to define the precedence order). These definitions were considered to create the algorithm that parses the AST to

obtain the elements to comprise the visual structure (Figure 1). At this point we are able to execute the three steps of the reference model presented in Figure 1. Now we can decide about the views to make available for the comprehension of the routines of the two languages. To accomplish this, we will present in the next section the concepts related to CCCs in MATLAB and Octave programs.

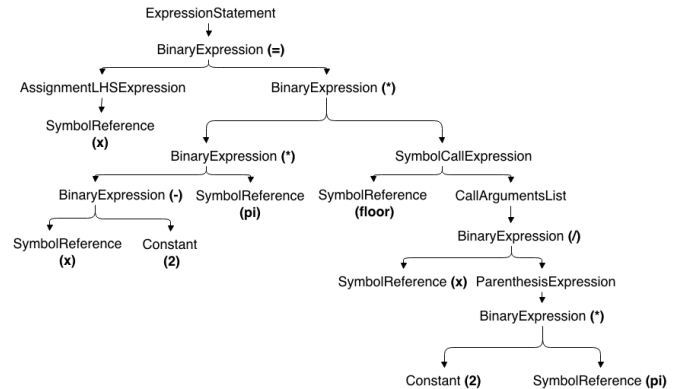


Figure 4: AST Structure of a MATLAB/Octave Program

V. CROSSCUTTING CONCERNS IN MATLAB AND OCTAVE

This subsection presents a scenario that illustrates comprehension tasks of MATLAB and Octave programs, how concerns are identified as well as how they are show up in the routine. A concern – or abstraction, or would-be module – is anything a stakeholder wishes to consider a conceptual unit, namely domain-specific features, nonfunctional requirements, and design patterns [10][13]. Some concerns are known to cut across module boundaries in systems developed by means of traditional technologies, including distribution, persistence, transaction management and security, which are found in many programs [4]. Such concerns are known as crosscutting concerns [6]. Studies have reported that acknowledging how concerns are manifested in the code supports programmers to better understand it as well as provides hints to the logic behind it [4]. Two primitive symptoms of the presence of CCCs in source code are scattering and tangling [10]. Scattering [10] is the degree to which a concern is spread over different modules or other units of decomposition. Tangling [13] is the degree to which concerns are intertwined to each other in the same functions.

Studies about concerns reported in the literature are focused on object-oriented systems in most cases, with an emphasis on the Java programming language [14]. Cardoso et al. [2] highlighted the importance of the identification and analysis of concerns in MATLAB. To this end, they analyzed 17 repositories and a total of 209 MATLAB programs to identify relevant concerns considering the number, local and frequency of their occurrences in the programs. They presented a list of concerns grouped in nine categories as shown in Table 1. Table 2 shows examples of MATLAB functions that illustrate each of these categories.

Monteiro et al. [7] proposed a simple but effective, token-based, approach for identifying and locating CCCs in a MATLAB repository. It is based on the idea that any

decomposition unit (function in the case of MATLAB) should ideally enclose a routine relating to a single concern [7]. However, in many cases, a given function encloses code that can conceptually be traced to more than one concern [7]. In light of this fact, a CCC in a function is always a secondary concern that is found in addition to the core concern. Discovering some of these concerns is not a trivial task. The authors identified a preliminary list of CCCs categories upon manual inspection of a number of real cases, complemented with insights acquired upon analysis of data collected from a specific tool. The core idea is to count occurrences of function calls in non-comment lines [7]. For each separate MATLAB file, several metrics are computed, including: i) number of times a given function name appears in a given MATLAB file and; ii) number of different functions appearing in a given MATLAB file [7]. Based on this scenario, the MVIE for MATLAB and Octave programs supports the identification of CCCs through the use of multiple views. According to previous studies, MVIEs provide effective support to that end [3][4].

VI. MAPPING REAL TO VISUAL ATTRIBUTES IN OCTMINER

To achieve the comprehension goals with the support of the multiple views, an important step is the mapping of real attributes (attributes from the original data properties domain such as MATLAB/Octave functions) to visual attributes such as shapes, colors and position in the screen. This step is the visual mapping conveyed in Figure 1, i.e. a transition from data to the visual forms [1]. Visual metaphors are instantiated to represent data stored in the visual structures. Moreover, they use the results of the mapping described in this section to produce the visual representations of the data selected by the programmer. Table 2 presents the mapping from real to visual attributes of two views of *OctMiner*: the Treemap and Grid views. Each pair of real-visual attribute is associated with a comprehension goal to be achieved while using the views. The results of these mappings can be seen in the views represented in figures 6-9. For example, the part G of Figure 6 presents the Treemap view while the part D of the same figure presents the Grid view. Besides the views created as a result of these mappings, the MVIE provides functionalities to interact with these views and to adjust them to best fit the comprehension goals. The next section describes an example to illustrate the functionalities.

VII. AN EXAMPLE OF USING OCTMINER

The example presented in this section aims at discussing the main functionalities provided by *OctMiner*. It is based on the study conducted by [7]. The main goal of the study was to identify the more frequently used tokens in the programs of a specific repository and how they are manifested in the routine in terms of tangling and scattering. The selected repository had 22 MATLAB files/programs. *OctMiner* was used to support the identification of tokens that were used most often in the repository. Figure 6 depicts a typical scenario of the MVIE integrated with the IDE Eclipse. Part A of the figure is the Project Explorer provided by the *Octclipse*.

Table 1. Concerns and their Categories in MATLAB and Octave [2]








Concern's Category	Category Description	Color Assigned in <i>OctMiner</i>	Examples of MATLAB and Octave Functions
Messages and monitoring	Messages to the user, warnings, errors, graphics visualization, monitoring, etc.;		plottools, semilogx, semilogy, loglog, plotyy, plot3, grid, title, xlabel,
I/O data	Reading data from file, writing data to file, saving an image, loading an image, etc.;		imwrite, imread, imformats, hgsave, saveas, hload, save,
Verification of function arguments and return values	Default shapes and values for the arguments that may not be passed in certain function calls;		nargchk, nargin, narginout, varargin, vararginout;
Data type verification and specialization	Check whether a variable is of certain type, configuring the assignment of data types to variables, etc.		quantize, quantizier, fi, isscalar, isstruct, iscell, isempty;
System	Code that verifies certain system environment properties, to pause execution, etc.	White	pause, print, printop, wait, last, input, syntax, run, tic, start;
Memory allocation and 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.;		clear, delete, zeros, persistent, global;
Parallelization	Use of parallel primitives such as "parfor";		parfor, spmd, gpuDevice, feval, demote, taskStartup, cancel, submit, resume;
Dynamic properties	Constructing inline function objects (inline), executing a string containing MATLAB expressions ('eval'), etc.		eval, evalc, evalin, inline

Table 2. Mapping Real to Visual Attributes in Two *OctMiner* Views

TREEMAP VIEW		
Real Attribute	Visual Attribute	Comprehension Goal
File	External Rectangle	Identify the different files analyzed in the repository
Token	Internal Rectangle	Identify the tokens used in each file
Number of Tokens per file	Size of rectangles	Identify the number of tokens per file
Category of each token	Color	Identify the category of crosscutting concerns (CCC) that a token belongs to
GRID VIEW		
Tokens	Rectangle	Identify the tokens used in a repository
Number of Tokens per file	Number in each rectangle	Identifies the number of tokens in each file
Category of tokens	Color	Identify the category of crosscutting concerns (CCC) that a token belongs to
LIST VIEW		
File	List of Files	Identify the name of the files and their location
Token	List of Tokens	Identify the name of the tokens and their location
Category of each token	Color	Identify the category of crosscutting concerns (CCC) that a token belongs to

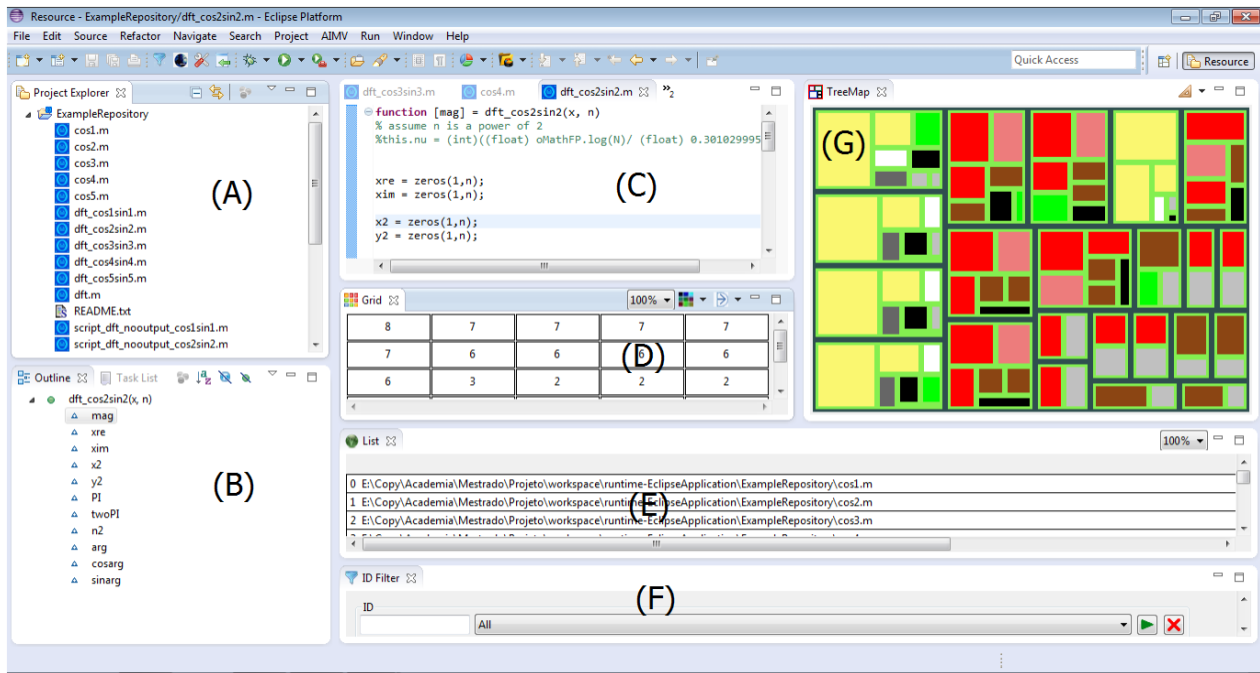


Figure 6. A Typical Scenario of *OctMiner* Use in the Eclipse IDE

This view presents all the repository files in different views (parts D, E and G of Figure 6). When one of these files is selected in the Project Explorer, their corresponding functions and variables are conveyed in the Outline (part B of Figure 6). The Outline is also provided by the *Octclipse* plugin. The corresponding routine of the same program is depicted in the Editor (Part C). The Grid (Part D), List (Part E) and Treemap (Part G) are the visual metaphors provided by *OctMiner* to support the comprehension of the programs in the repository. In the case of the Treemap view, it is possible to spot in a panoramic fashion how the tokens are distributed in the repository among the 22 files (Part G). Analyzing the colors presented in the Treemap view it is possible to conclude that the more representative tokens of the repository are classified in the following categories: dynamic properties (yellow), data type verification and specialization (red), design space exploration (brown), memory allocation/deallocation (black) and verification of functions arguments, i/o data (light green) and returns values (light gray). Passing the mouse in each rectangle it is possible to identify the name of each token using the tooltip resource provided by the view.

The goal of the List view (part E of Figure 6) is to present a list of the programs of the repository and to enable the selection of one of them to be highlighted in the other views. The goal of the Grid view is to represent the programs with their respective number of types of tokens manifested in the routine enabling the user to see the programs both in ascending or descending order according to the number of types of tokens. To identify the most used token in the repository the user can change the filter dataset in *OctMiner* from “All” to “Tokens”. At this point, the new visual scenario corresponds to the one depicted in Figure 7.

In Figure 7 all the types of tokens manifested in the repository are visually represented. All three views have their

visual entities colored indicating the category that a token belongs to. The List view now presents the list of types of tokens colored. The Treemap view presents the quantity of types of tokens in the repository. The grid view presents the quantity of these tokens in all the programs of the repository. Together, these views allow the user to identify the tokens that are used most often in the repository and their corresponding categories.

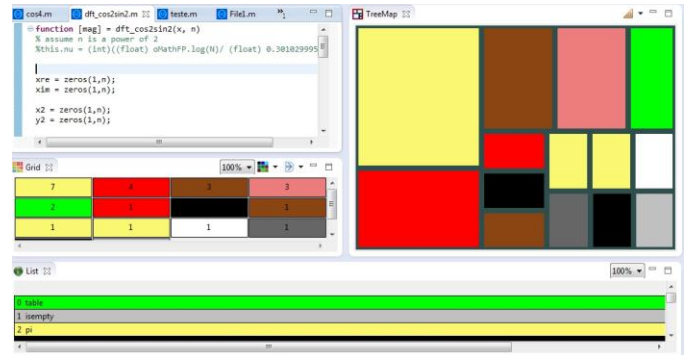


Figure 7. *OctMiner* Views Focusing on the Tokens Representation

This visual scenario enables the analysis of how a specific token is manifested in the repository. To accomplish this, the user should use the filter ID to select the token to be highlighted in the views. In the example of Figure 8, the token “signal” is selected. As can be seen in the figure, the red arrows marked in the figure indicate the occurrence of this token in each program. The yellow color stands for the category of token “dynamic properties”. The size of each rectangle represents the proportion of use of this token in the file compared with the others. This is an interesting view for

the analysis of CCCs and for further conclusions about their symptoms in terms of scattering and tangling.

The views provided by *OctMiner* enable the analysis of how the token “signal” is scattered in the repository files and to which category it belongs to. It is also possible to apply the semantic zoom to navigate over a specific data set from a panoramic view. Figure 9 presents how tokens are manifested in a given program. In this case, the program is the lowest level of abstraction visually represented in the views.

This level of detail enables the visualization of each program presented in the List view, how the tokens are manifested using the Treemap view and the quantity of tokens by analyzing the Grid view.

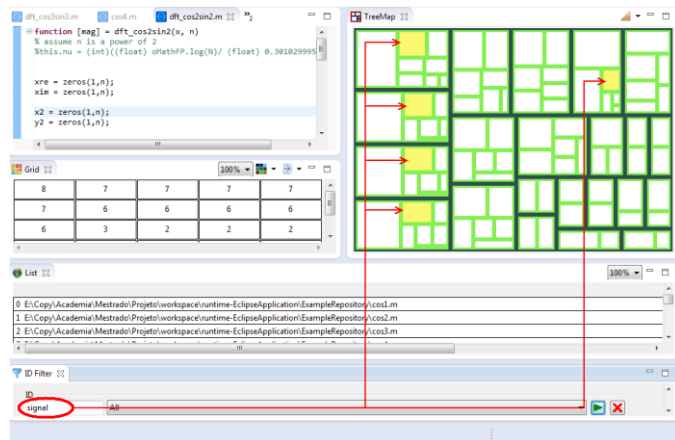


Figure 8. *OctMiner* Views Focusing on the All “signal” Representation

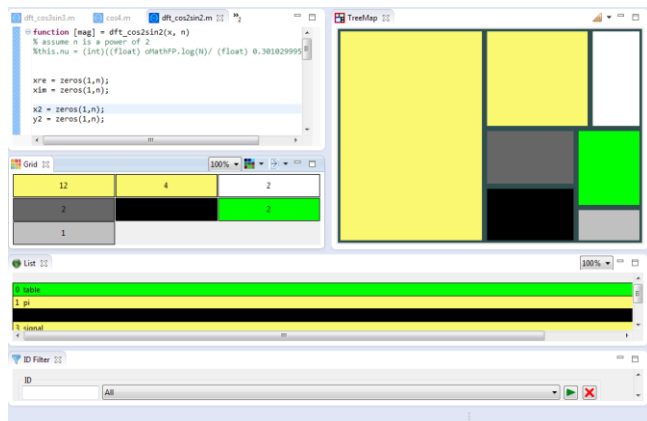


Figure 9. *OctMiner* Tailored to Present Tokens in a File

These evidences let us conclude that *OctMiner* can support the comprehension goals discussed [7]. A controlled experiment is being planned by the authors to analyze this effectiveness in more details and based on both quantitative and qualitative analysis.

VIII. CONCLUSIONS AND FUTURE WORKS

This paper presented *OctMiner* as a multiple view interactive environment to support the comprehension of MATLAB/Octave programs. Multiple views are tailored to represent data according to the comprehension goals discussed in [7].

The paper also presented an architectural overview of *OctMiner* plugin integrated on the Eclipse IDE. *OctMiner* works upon data provided by the AST of MATLAB and Octave programs so that these data can be transformed to appropriated visual data structures to feed the views that comprise the multiple view environment. As future work, we are planning a controlled experiment with a repository with a large number of MATLAB and Octave programs to identify the more referenced tokens as indicators of the presence of CCCs, as well as their symptoms in terms of scattering and tangling more thoroughly.

REFERENCES

- [1] Card, S. K., Mackinlay, J. and Shneiderman, B. Readings in Information Visualization Using Vision to Think. San Francisco, CA, Morgan Kaufmann, 1999.
- [2] Cardoso, J.; Fernandes, J; Monteiro, M.; Carvalho, T; Nobre, R. Enriching MATLAB with aspect-oriented features for developing embedded systems. Journal of Systems Architecture 59 (2013) p. 412–428.
- [3] Carneiro, G.; Mendonça, M.. SourceMiner: Towards an Extensible Multi-perspective Software Visualization Environment. In: Slimane Hammoudi; José Cordeiro; Leszek A. Maciaszek; Joaquim Filipe. (Org.). Enterprise Information Systems. 1ed.: Springer International Publishing, 2014, v. 190, p. 242-263.
- [4] Carneiro, G., Silva, M., Mara, L., Figueiredo, E., Sant’Anna, C., Garcia, A., Mendonça, M., 2010. Identifying code smells with multiple concern views. In: XXIV Brazilian Symp. on Software Engineering (SBES 2010), IEEE Comp. Soc., Washington, DC, USA, pp. 128–137.
- [5] Chaves, J.; Nehrbass, J.; Guilfoos, B.; Gardiner, J.; Ahalt, S.; Krishnamurthy, A.; Unpingco, J., Chalker, A.; Warnock, A.; Samsi, S. Octave and Python: High-Level Scripting Languages Productivity and Performance Evaluation. In Proc. of the HPCMP Users Group Conference (HPCMP-UGC ’06).
- [6] Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C., Loingtier J., Irwin J., Aspect-Oriented Programming. ECOOP’97, Jyväskylä, Finland, June 1997.
- [7] Monteiro, M.; Cardoso, J.; Posea, S. Identification and characterization of crosscutting concerns in MATLAB systems. In Conference on Compilers, Programming Languages, Related Technologies and Applications (CoRTA 2010), Braga, Portugal (pp. 9-10).
- [8] Nunes, A.; Carneiro, G.; David, J. Towards the Development of a Framework for Multiple View Interactive Environments. In: International Conference on Information Technology: New Generations (ITNG), 2014, Las Vegas/EUA. p. 23-30.
- [9] Octave Programming Language. Available at www.gnu.org/software/octave/.
- [10] Robillard, M; Murphy, G. Representing Concerns in Source Code. ACM TOSEM, 2007.
- [11] Spence, R. Information Visualization: Design for Interaction (2nd Edition). 2. ed. Prentice Hall, 2007.
- [12] Stenroos, M.; Mäntynen, V.; Nenonen, J. A MATLAB library for solving quasi-static volume conduction problems using the boundary element method. - Computer methods and programs in biomedicine, 2007.
- [13] Tarr, P.; Oshser, H.; Harrison, W., Jr., N. Degrees of Separation: Multi-Dimensional Separation of Concerns. ICSE, 1999.
- [14] Uirá Kulesza et al. The crosscutting impact of the AOSD Brazilian research community. J. Syst. Softw. 86, 4 (April 2013), 905-933.
- [15] Walid Maalej, Rebecca Tiarks, Tobias Roehm, and Rainer Koschke. 2014. On the Comprehension of Program Comprehension. ACM Trans. Softw. Eng. Methodol. 23, 4, Article 31 (September 2014), 37 pages.