A Parser and a Software Visualization Environment to Support the Comprehension of MATLAB/Octave Programs

Thiago de Lima Mariano[1], Glauco de Figueiredo Carneiro[1], Miguel Pessoa Monteiro[2], Fernando Brito e Abreu[3] and Ethan Munson[4]

[1] *Universidade Salvador (UNIFACS), Salvador, Bahia, Brazil*
[2] *Universidade Nova de Lisboa (NOVALINKS), Lisbon, Portugal*
[3] *Instituto Universitario de Lisboa (ISCTE-IUL), Lisbon, Portugal*
[4] *University of Wisconsin Milwaukee (UWM), Milwaukee, Wisconsin, U.S.A.*

Keywords: MATLAB/Octave, Parser, Abstract Syntax Tree (AST), Knowledge Discovery Metamodel (KDM), Software Comprehension, Software Visualization.

Abstract: Software comprehension and analysis of MATLAB and Octave programs are not trivial tasks. Programmers have to devote considerable effort to obtain relevant data from source code and related artifacts. Tools that provide support for software comprehension activities usually rely on parsers to obtain data from source code. The problem in the MATLAB/Octave case is the limited number of available parsers and the difficult to build an extensible solution with them. In this paper, we describe the development of a parser that converts MATLAB and Octave program codes into instances of the Knowledge Discovery Metamodel (KDM), which can subsequently undergo static analyses to feed different visual representations. The goal of these representations is to support software comprehension. We describe our experience in the use of this parser to build a software visualization environment to support the comprehension of MATLAB and Octave programs.

1 INTRODUCTION

Software visualization has supported programmers and designers to discover information that are hidden in standard parsing processes (Munzner, 2014)(Seriai et al., 2014). In this paper, we focus on the parser implementation and a visualization environment for two programming languages. The first is MATLAB, a very popular dynamic scripting language for numerical computation and matrix processing used by scientists, engineers and students (Radpour et al., 2013). The second is Octave [1], a (mostly compatible) open source MATLAB clone. Both languages provide resources for solving common numerical linear algebra problems, e.g., computing the roots of nonlinear equations, integrating ordinary functions, manipulating polynomials, as well as integrating ordinary differential and differential-algebraic equations. In most cases, programs written in those languages are often developed incrementally using a combination of scripts and functions and frequently build upon existing code.

[1]https://www.gnu.org/software/octave/about.html

The execution of Octave programs uses its own interpreter that is available in the project website [2]. This interpreter is a *parse-tree* library developed in C++ that generates the code's *Abstract Syntax Tree* (AST). Although it covers all the features of Octave, it does not include documentation or references regarding its source code. Comprehension of the parser is not trivial, especially considering the effort to understand the attributes and methods. To illustrate, some methods found in the parser source code include: `yypull_parse`, `yypstate_new`, `yyparse`, `yyalloc` and `yytnamerr`. The MATLAB interpreter also includes a parser, but it is not possible to verify the type of output generated, since this language is proprietary and does not make its source code publicly available. Our initial aim was to use one of the options mentioned above, but this did not prove feasible due to the aforementioned limitations. Considering the variations between the MATLAB and Octave languages, adjustments must be done to the source code of the parse-tree to cover the programs developed in both programming languages.

[2]https://ftp.gnu.org/gnu/octave/

de Lima Mariano, T., de Figueiredo Carneiro, G., Pessoa Monteiro, M., Brito e Abreu, F. and Munson, E.
A Parser and a Software Visualization Environment to Support the Comprehension of MATLAB/Octave Programs.
DOI: 10.5220/0006644701790186
In *Proceedings of the 20th International Conference on Enterprise Information Systems (ICEIS 2018)*, pages 179-186
ISBN: 978-989-758-298-1

We have identified a set of parsers developed for the MATLAB and Octave programming languages: *Octclipse*[3], *GNU Octave*[4], *ANTLRv4*[5], *McSAF* (Doherty and Hendren, 2012), Magica tool (Joisha and Banerjee, 2003) and MaJIC (Almási and Padua, 2002). However, we have observed that they are not suitable nor provide appropriate conditions to be extended and adapted to provide data to software comprehension tools. For example, none of them produce or export KDM representations to be further used in such tools. *Octclipse* is an Eclipse plugin that supports the development of Octave applications. It has no development activity since 2015 and according to its documentation, is compatible to Octave only until version 3.6. Therefore, some features found in more recent versions of Octave are not covered by its corresponding parser. Similar limitations are reported in the *ANTLRv4* documentation, as regards the capability of generating the AST from MATLAB programs. Updates to this parser were not made since 2015 and likewise does not support the more recent language features. In addition, the parser does not cover features such as *switch statements*, *structures* and *cell arrays*. *McSAF* is a compiler analysis framework that is intended to enable compiler research by providing both an intermediate representation and an intraprocedural analysis framework which can be used both for MATLAB and language that are extensions of MATLAB (Doherty and Hendren, 2012). However, the main limitation of *McSAF* is the significant effort it requires to produce or export KDM representations.

The *Magica* tool (Joisha and Banerjee, 2003) deals with type inference for matrix operations and functions. In addition to inferring the intrinsic type of matrices, such as int32, double, or char, it also infers matrix sizes and shapes. *Magica* has the goal to perform code optimization. Another compiler project for MATLAB is *MaJIC* (Almási and Padua, 2002). It uses a Just-In-Time (JIT) compiler component to achieve speedups. The main difference of focus between these two projects and the parser presented here is that their main goal was to improve the performance of MATLAB programs. They do not have the main goal to provide data to tools that support the comprehension of programs coded in the two aforementioned languages.

Although tools are available to support the development of MATLAB and Octave applications, to the best of our knowledge just a few seem to be geared to source code analysis and comprehension. We could perhaps mention *MATLAB Online* [6], *Octave Online* [7], *JDoodle* [8] and *Coding Ground* [9]. *MATLAB Online* is a web version of the Mathworks software development tool. In this release, source code files are stored in *MATLAB Drive*. *Octave Online* is a web tool used in the development of Octave applications. *JDoodle* and *Coding Ground* are web tools for developing applications in various programming languages, including MATLAB and Octave. However, these tools listed before mainly provide an online programming environment where programmers can type and run their code. They lack sophisticated support for software comprehension and analysis such as the use of visualization resources to identify the entities that comprise code and their relationship (Storey, 2006) or traceability between functional requirements of the program and the correspondent piece of code that implements it (Chen, 2010).

In previous work, we implemented an Eclipse plug-in to support MATLAB and Octave programs comprehension through visualization resources (Lessa et al., 2015b)(Lessa et al., 2015a). The plug-in relied on a parser provided by the GNU Octave project. The tool was dependent on Eclipse IDE and it was not possible to export or import data from/to the tool. Moreover, the visual metaphors that represented code entities and possible relationships between them were implemented using the Draw2d layout and rendering toolkit building on top of SWT in Eclipse (The Standard Widget Toolkit), which does not provide visualization resources such as those used by some web pages currently available. From the users perspective, this has an impact on the attractiveness of the visual metaphors.

The rest of this paper is organized as follows. Section 2 presents the parser implemented to convert MATLAB/Octave source code to instances of the Knowledge Discovery Metamodel (KDM). Section 3 describes how the proposed parser was integrated to a visual interactive environment aimed at supporting the comprehension of MATLAB/Octave programs. Section 4 conclusions and directions of future work.

2 PARSER IMPLEMENTATION

The goal of the proposed parser is to support different kinds of tasks, including the provisioning of data to

[3]https://sourceforge.net/projects/octclipse/

[4]https://www.gnu.org/software/octave/

[5]https://github.com/ericharley/matlab-parser

[6]https://www.mathworks.com/products/matlab-online.html

[7]https://octave-online.net/

[8]https://www.jdoodle.com/execute-octave-matlab-online

[9]https://www.tutorialspoint.com/codingground.htm

feed visual metaphors to represent MATLAB/Octave programs. The literature has already called attention to this need. For example, in (Monteiro et al., 2010), the authors provide a short review of the state-of-the-art of aspect mining and particularly the identification and localization of crosscutting concerns (CCCs) as latent aspects. They conclude that until that time, not enough attention was paid to this topic in the context of MATLAB systems.

In the context of compilers in programming languages, a parser aims at analyzing the given sequence of tokens to build a hierarchical data structure according to previously established syntactic rules. In most cases, the output of this transformation is a tree structure, whose building process is carried out either through top-down or bottom-up approaches. In the first case, the tree is built from the root to the leaf nodes, while in the bottom-up approach the flow is in the opposite way (Aho et al., 2007). One of the main advantages of the parser presented in this paper is the conversion of source code of MATLAB/Octave programs to KDM instances that can be subsequently used to provide data to support the analysis and comprehension of target systems. In our case, we use the data converted by the proposed parser to feed a software comprehension visualization environment.

Parser Steps. Having this in mind, we developed the parser in two steps. Step 1 comprises an intermediate parser capable of generating the AST of MATLAB/Octave programs from its source code (***parts A, B** and **C*** of Figure 1). Step 2 comprises another partial parser to generate KDM instances from AST (***parts C, D** and **E*** of Figure 1). In this case, ***part E*** shows the data to be provided to software comprehension tools to support program analysis.

To deal with several small differences between MATLAB and Octave, we performed a prior analysis of their grammars (Version 4.2.0 of the GNU Octave) [10], the packages that comprise the KDM metamodel and how the KDM metamodel can effectively represent MATLAB/Octave programs. This is a key issue for the proposed parser presented in this paper.

During the language analysis phase (***part A*** of Figure 1), the set of operations is identified, how they work and which operators can be used. This includes arithmetic operators, comparison operators, boolean expressions, function calls, assignment expressions, increment operators, and role definitions. Moreover, several control structures (e.g., `if`, `switch`, `while`, etc) and statements such as `continue`, `break`, `unwind`, `protect`, `try` and continuation lines are also considered. The `ASTOctave` library recognizes each of these operations, creates an element to represent them and registers them as an AST node (***part C*** of the Figure 1). The version 4.2.0 of the GNU Octave documentation[11] was used as a support for this implementation.

The analysis of the KDM metamodel was a requirement to implement ***part D*** of Figure 1. More specifically, we identified twelve packages from the KDM metamodel whose elements aimed at representing a specific part of the system under analysis. Packages from the KDM metamodel are defined in layers. The `Code` package is the main package upon which all the others provide their respective services. In the `Parser` implementation described here, we use elements from the five lower packages: `Core`, `kdm`, `Source`, `Code` and `Action`. These elements where then selected to be part of the XMI file to describe the target system.

Considering all the steps presented in Figure 1 to convert MATLAB/Octave programs to KDM instances, we implemented the following five libraries `ASTOctave`, `ASTOctaveToKDM`, `OctaveKDMStructure`, `OctaveKDMToJSON` and `OctaveKDMToXMI`.

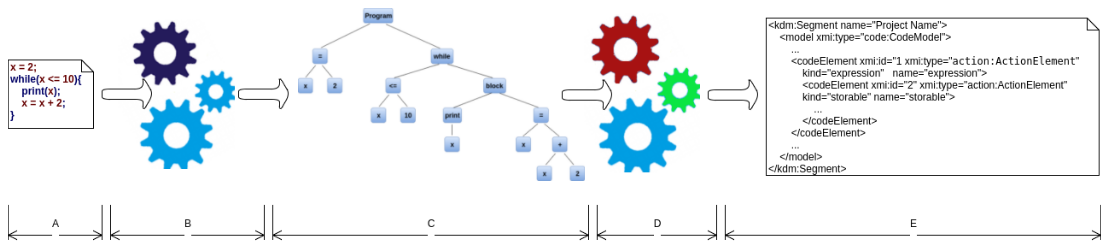
The first library of the pipeline is `ASTOctaveToKDM` whose goal is to scan all the source files from an Octave or MATLAB target project. Library `ASTOctaveToKDM` is the central mediator of the proposed set of libraries, being responsible for instantiating the KDM representation of the project to be parsed. It is also responsible for controlling when and how other libraries are instantiated and used. In this scenario, it instantiates and calls library `ASTOctave` aimed at reading a MATLAB or Octave file and generating the corresponding AST. Library `ASTOctave` also creates the project's KDM metamodel. Library `OctaveKDMStructure` represents the KDM structure of MATLAB and Octave applications.

KDM Packages Description (OMG, 2016). **Core:** Defines the basic elements to be used by all other packages, including the entities, their relationships, and the basic data types, such as `String`, `Boolean`, and `Integer`. Therefore all other packages depend on it but `Core` does not depend on any of them. These basic data types are used to represent the KDM attributes, their operations, and their parameters.

KDM: Defines key environment elements setting the standards for building KDM views on the target system. The elements described in this package, together with the elements described in *Core*, comprise the KDM framework.

[10]www.gnu.org/software/octave/doc/v4.2.0/index.html#SEC Contents

[11]www.gnu.org/software/octave/doc/v4.2.0/index.html#SEC_Contents



```
x = 2;
while(x <= 10){
   print(x);
   x = x + 2;
}
```

```
<kdm:Segment name="Project Name">
   <model xmi:type="code:CodeModel">
      ...
      <codeElement xmi:id="1 xmi:type="action:ActionElement"
         kind="expression"   name="expression">
         <codeElement xmi:id="2" xmi:type="action:ActionElement"
         kind="storable" name="storable">
            ...
         </codeElement>
      </codeElement>
      ...
   </model>
</kdm:Segment>
```

Figure 1: Parser Steps: from the Source Code to the KDM Instance.

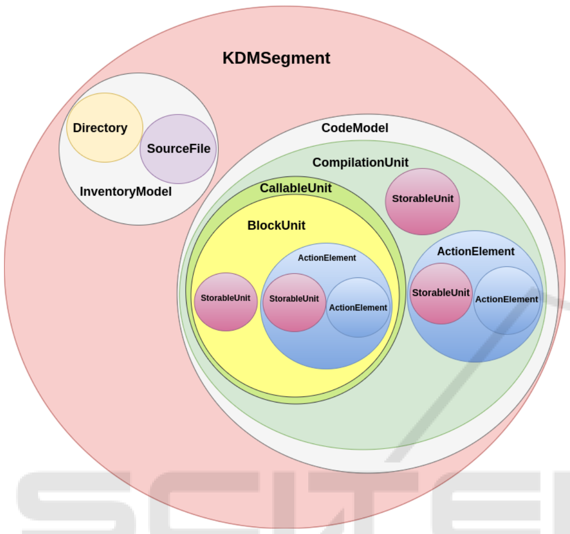


Figure 2: Hierarchical structure of the KDM objects.

Source: Defines elements that are used to represent the physical artifacts of existing software, e.g., images, source files, configuration files. For instance, if the target system has ten files, each file will be represented by an element defined in `Source`.

Code: Has an extensive set of elements used to represent the implementation of the target system. Each instance of a `Code` element corresponds to a region of the source code of an artifact from the target system. **Action:** Defines a set of elements representing behavior determined by programming languages, e.g., declarations, operators, conditions.

The KDM elements used to represent MATLAB/Octave programs, with their respective hierarchy, is represented in Figure 2. Finally, libraries `OctaveKDMToXMI` and `OctaveKDMToJSON` scan the data stored under the project's KDM structure and respectively generate the XMI and JSON files of that structure. We decided to use the JSON format for the parser output, to allow native manipulation of the data using *NoSQL* database. The goal is to facilitate information retrieval from the data stored in *NoSQL* databases. This was the motivation for developing library `OctaveKDMToJSON`.

Parser Execution. The execution of the proposed parser entails using library `ASTOctaveToKDM` as the central point and reference for the other libraries. The method `parseProjectToKDM` is responsible for creating the KDM structure to be located in the path indicated as a parameter.

After the process is initiated, library `ASTOctaveToKDM` traverses all the project's source code files. For each file found, it calls library `ASTOctave` through method `generateAST`, passing the file as parameter. Library `ASTOctave` reads the contents of the file and generates the corresponding AST, returning it to library `ASTOctaveToKDM`. Library `ASTOctaveToKDM` scans all nodes and converts each AST element into a KDM element through the use of library `OctaveKDMStructure`. When all within AST nodes are converted, library `ASTOctaveToKDM` passes the KDM instance to libraries `OctaveKDMToXMI` and `OctaveKDMToJSON`. In fact, these instances represent the project under analysis. The last mentioned two libraries (`OctaveKDMToXMI` and `OctaveKDMToJSON`) read all KDM elements and respectively generate the corresponding XMI and JSON files.

3 BUILDING A SOFTWARE VISUALIZATION ENVIRONMENT

In this section, we report on the experience gained during development of the software visualization environment -whose main goal is to support the comprehension of MATLAB/Octave programs based on visualization resources using data provided by the parser. Its architecture is presented in Figure 4, which highlights the role of the proposed parser to convert data from ***part A*** to part ***B*** of the Figure.

The technologic solution is comprised of the *Apache Tomcat* web container, the *CouchDB NoSQL* database management system, the REST Java API to create the services, the *jQuery Ajax*, and *D3.js*, a Java Script library to manipulate DOM objects based on JSON data. These technologies are used to implement parts ***B***, ***C***, ***D***, and ***E*** of the visualization pipeline shown in Figure 4.

3.1 The Persistence Level

One of the main reasons for selecting a *NoSQL* document oriented database is its native flexible schema functionality. As a result, two or more documents can have very different schema and data values, unlike in the relational model, where each row in a table will have same columns (Li and Manoharan, 2013). For example, data related to different MATLAB and Octave programs can be stored in a document-oriented database where each document can contain the software entities of a specific file and their respective relationship. Each software entity has a set of attributes that may vary from one entity to another. In that case, a relational model would not be suitable to deal with this variation.

Considering the four options of *NoSQL* database templates available, we decided to use the model that best fits the way data is stored within the application source code is document-oriented. The suitability of the data structure of document-oriented databases to the data structure of the JSON files (a tree, where a code block can have one or more blocks inside it in a nested form) created by the parser presented in this paper influenced the adoption of a document-oriented database.

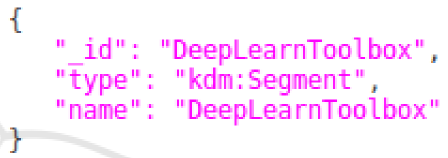
We decided to adopt the document-oriented NoSQL *CouchDB* database (*Cluster of Unreliable Commodity Hardware*[12]) for the environment, as it is a distributed schema database providing a *ResTful JSON* (*Java Script Object Notation*) to manage documents stored through HTTP calls. The *CouchDB*-based HTTP mechanism is scalable to heavy loads of requests by responding to HTTP calls using the `GET`, `POST`, `PUT`, or `DELETE` http methods (Anderson et al., 2010).

3.2 Parsing Raw Data

Part A of the environment described in Figure 4 corresponds to the raw data - the code of the projects to be analyzed in this case. We chose the Github repository, mainly due to the number of MATLAB and Octave designs available. In addition, Github is a world-renowned public repository, widely used and respected by free software developers. ***Part B*** of the environment from Figure 4 represents the transformation from code to data structures - to be stored in a database - performed by the parser. See section 2 for details on the transformation steps of the original data (program code).

[12]https://goo.gl/mtnk

The data is stored in the database as KDM elements, which represent the entities that comprise the projects and their respective information, such as their relationship with other elements of the program. We needed to map these elements to documents in the JSON format, a task that was also carried out by the parser. The resulting JSON files are illustrated in ***part B*** of Figure 4. The first element found in the KDM structure is the *KDMSegment* object, which represents the project itself and is converted into a JSON document with the following attributes: *id*, *type* and *name*. In attributes *id* and *name*, the project name was entered and the *String* ”*kdm:Segment*” was inserted into attribute *type*. The attribute of all other documents is generated automatically by the database. Figure 3 shows a section of this document.



```
{
    "_id": "DeepLearnToolbox",
    "type": "kdm:Segment",
    "name": "DeepLearnToolbox"
}
```

Figure 3: A JSON Document Sample Representing the KDMSegment Object.

The next element found is object *CodeModel*. In addition to the attributes created in the *KDMSegment*, two new attributes are created: *idParent* and *idProject*, which will have the same value corresponding to attribute *_id* of object *KDMSegment*. The same goes for object *InventoryModel*. Attribute *type* entered in the document that represents the *CodeModel* receives as value the *String ”code:CodeModel”*. In the case of the *InventoryModel* object, the *String ”source:InventoryModel”* is assigned to the attribute *type*. All other documents have the *idProject* attribute and the *id* of the document that represent the *KDMSegment* is added to this attribute.

InventoryModel objects can contain *Directory* and *SourceFile* objects. It is a part of the KDM instance environment. As a general rule, each instance of the inventory model represents a file or a set of files in a given KDM instance.

This meta-model element is a container of instances of other inventory meta-model elements. In addition to the attributes already mentioned, the documents that represent these objects also have the attributes *path* and *language*. The *String ”MATLAB/Octave”* is assigned to the *language* attribute, whereas the *path* attribute receives the directory path in which these files or folders are located within the project. The *type* attribute of the *SourceFile* object receives the *String ”source:SourceFile”*. In the case

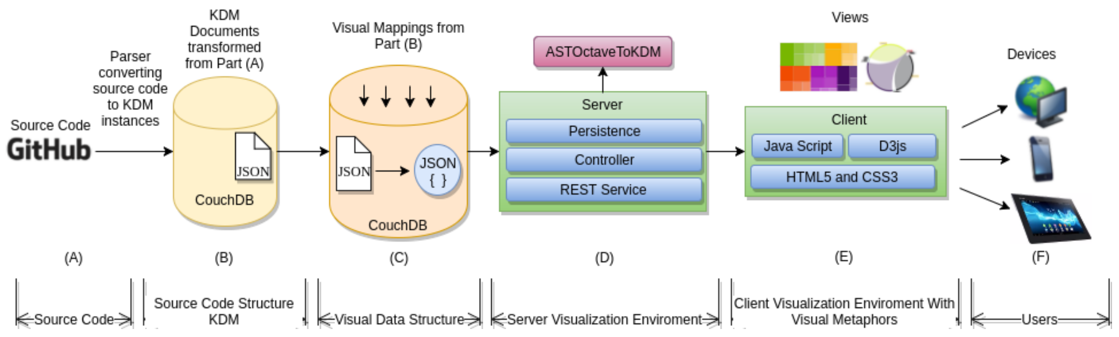

Source Code
GitHub
Parser converting source code to KDM instances
KDM Documents transformed from Part (A)
JSON
CouchDB
Visual Mappings from Part (B)
JSON
JSON { }
CouchDB
ASTOctaveToKDM
Server
Persistence
Controller
REST Service
Views
Client
Java Script
D3js
HTML5 and CSS3
Devices
(A)
(B)
(C)
(D)
(E)
(F)
Source Code
Source Code Structure KDM
Visual Data Structure
Server Visualization Enviroment
Client Visualization Enviroment With Visual Metaphors
Users

Figure 4: Environment to Support the Visual Analysis of MATLAB/Octave Programs.

of the *Directory* object, the *type* attribute receives the *String "source:Directory"*. The *idParent* attribute of these documents receives the *_id* of the document that represents the *InventoryModel* object.

Each *CompilationUnit* object found in the *CodeModel* object is converted to a different JSON document with the attributes *id*, *name*, *type*, *idProject*, and *idParent*. The *name* attribute is given the name of the source file that this element represents. The *type* attribute receives the *String "code:CompilationUnit"* and *idParent* receives the id of the document that represents the *CodeModel* element. We decided to convert the *CodeModel* object to different JSON documents due to performance reasons. The memory storage capacity of *CouchDB* does not accept a single document containing multiple *CodeModel* objects. We observed that carrying out queries targeting multiple documents, one for each object, also provided better performance results. Element *CallableUnit* is converted into a document with attributes *_id*, *type*, *name*, *idParent*, *idProject*, *idCompilation* and *compilation*. Attribute *type* receives the *String "code:CallableUnit"*. Attribute *name* is given the name of the function represented by this element. Attribute *idParent* receives the id of the document that represents the *CompilationUnit* to which this function belongs. Attribute *idCompilation* receives the id of the document that represents the *CompilationUnit* element to which this function is part and attribute *compilation* is given the name of the file itself.

The documents representing the *Blockunit*, *CodeElement* and *StorableUnit* elements have attributes *_id*, *kind*, *name*, *idParent* , *idProject*, *idCompilation*, and *compilation*. These documents can also receive attributes *idCallable* and *nameCallable*, if the elements they represent are inside an element that represents a function. Attribute *type* of the documents that representing elements *BlockUnit*, *CodeElement* and *StorableUnit* respectively receive values *"action:BlockUnit"*, *"action:ActionElement"* and *"code:StorableUnit"*. Attribute *kind* is populated with the value stored in variable *kind* of the KDM element. The same thing happens with attribute *name*. The rule for filling in the remaining attributes is exactly the same as that used in the document representing element *CallableUnit*.

Figure 5 shows an example of a JSON file representing the *CallableUnit* element of the KDM of a MATLAB/Octave program. Attributes *id* and *rev* correspond to the unique identifier of this document in the database. They are generated automatically by CouchDB. The *rev* attribute indicates the version of the document. The same document may have multiple versions in the *CouchDB*. The value assigned to attribute *type* signals that this element matches the function within a MATLAB/Octave program. Attribute *name* has the name of this function. Attribute *idParent* indicates the *id* of the parent document of that element representing a function. In this case, the parent document represents the *BlockUnit* element of the KDM. This *BlockUnit* corresponds to the code block of a program code file. Attribute *idProject* has the *id* of the document that represents the *KDMSegment* element of the KDM, represented in Figure 3. This element identifies the project under analysis. Attributes *idCompilation* and *compilation* identify the code file that contains this function. The KDM element that represents a code file is *CompilationUnit*. Attribute *idCompilation* has the id of the document that represents *CompilationUnit* and attribute *compilation* stores the name of the code file. Attribute *signature* stores the signature of a MATLAB/Octave function along with its parameters.

Through Figures 3 and 5, it is possible to verify that, in the KDM structure, the *CallableUnit* element represented in Figure 5 is hierarchically below element *BlockUnit*. Element *BlockUnit* is hierarchically below element *CompilationUnit*. Element *CompilationUnit* is hierarchically below element *CodeModel* and element *CodeModel* is hierarchically below element *KDMSegment* represented in Figure 3. This verification can be verified by linking attributes *idParent* and *id* of each of these elements.

To find information about MATLAB/Octave projects stored inside DBMS, *CouchDB* provides a feature for accessing data called *view*. Views are created within *CouchDB* itself as a JSON document and are defined through Java Script functions. Figure 6 represents a *view* created inside the *CouchDB*. This *view* is able to return all documents stored in the database whose attribute *type* has the value *"kdm:Segment"*.

```
{
  "_id": "c91436ca676f464d9abfbdecabc947d2",
  "_rev": "1-3af9e81243b7ceca927aa6f36e7d135d",
  "type": "code:CallableUnit",
  "name": "caebbp",
  "idParent": "d5b706860a3e49949f12fb330b515cf6",
  "idProject": "DeepLearnToolbox",
  "idCompilation": "1dacc19bfe304f5fbc663ca0c8ef5198",
  "compilation": "caebbp.m",
  "signature": {
      "type": "code:Signature",
      "name": "caebbp",
      "parameterUnit": [
          {
              "kind": "return",
              "name": "cae",
              "pos": "0"
          },
          {
              "kind": "parameter",
              "name": "cae",
              "pos": "1"
          }
      ]
  }
}
```

Figure 5: Data Structure of a JSON File Representing a KDM CallableUnit Element.

```
function(doc) {
  if(doc.type == "kdm:Segment"){
    emit(doc.id, doc);
  }
}
```

Figure 6: CouchDB View Sample.

The *view* displayed in Figure 6 was executed through the *Futon* tool, which corresponds to a web application used to manage *CouchDB*. This tool allows one to create databases, documents and *views*. In addition, these can be modified or removed. It is also possible to manage the *CouchDB* settings through *Futon*. This *view* functionality of *CouchDB* was used to loop through the set of documents that make up the KDM structure of a given project and mount a JSON file with the information needed for visualization. This process of transforming the original data into a new file in the format required for visualization is illustrated in ***part C*** of the environment described in Figure 4.

The Server Side of the Environment. The server side of the application, represented in part D of Figure 4, consists of Java classes that are responsible for containing the business rules of the application, the access to the *CouchDB* database and the services to access the data. It is also the responsibility of the server to use the library *ASTOctaveToKDM* to generate the KDM metamodel of the MATLAB/Octave programs and store it in the database.

We developed some methods in the server application controller that are responsible for traversing all the elements contained in the created KDM structure and converting them into JSON documents, which were subsequently saved in *CouchDB*. After mapping the KDM elements in the corresponding JSON document, a feature was developed to store it in the database through the use of the Java library *Lightcouch*, which corresponds to a Java Application Programming Interface (API) for communication with the *CouchDB* database.

The Client Side of the Environment: Implementing the Visualization Resources. The client side of the view environment represented in ***part E*** of Figure 4 corresponds precisely to the joining of the *html* pages with their Java Script functions and style sheets. It is at this stage that the visual metaphors, containing the MATLAB/Octave program data, are displayed. The visual metaphors available in the environment use the *D3.js*, which is an open source Java Script library that provides resources for manipulating HTML documents. It uses Java Script as the language for implementing data mapping for documents (Bostock et al., 2011).

The first visual metaphor indicated to be used in the proposed environment is *Treemap* [13]. This visual metaphor is capable of representing large volumes of hierarchical data through recursively nested rectangles representing domain-relevant entities also known as real attributes. Examples of these attributes are the number of source lines, the source code files, their respective functions, and other real attributes that can be visually represented by visual attributes such as rectangle, color, and size (Shneiderman, 1992). The second visual metaphor indicated to be used in the proposed environment is *chord* [14]. The *chord* view is capable of displaying a set of arcs forming a circle and a set of lines connecting each of these arcs to represent their relationships. The size of the arcs in the circle varies according to the number of connections you have with the other arcs of the circumference. While a small amount of data could be represented in a circular diagram using straight lines to show interconnections, a diagram with numerous lines would quickly become unreadable. To reduce visual complexity, chord

[13]https://bl.ocks.org/mbostock/4063582

[14]http://bl.ocks.org/NPashaP/ba4c802d5ef68f70c019a9706f77ebf1

diagrams employ a technique called *hierarchical edge grouping* (Holten, 2006). The *chord* view has been mapped to contain in each arc of the circumference a source code file of a MATLAB/Octave project and the lines thet connecting the arcs represent the afferent and efferent couplings between each of these files.

The communication between the client side of the application and the server side is done through of calls by the *Java Script* functions to services available on the server side by the *REST* technology. Rest is an architectural style that allows the development of services that can be accessed via calls by *url*.

4 CONCLUSION AND FUTURE WORK

The main goal of the parser is to generate KDM instances of programs developed in languages MATLAB/Octave, which in turn can be used as input to support processes of analysis and understanding.

In short, the advantages of the proposed parser in comparison with the related work are summarized as follows. The proposed parser convert MATLAB/Octave programs in KDM instances and these instances can be manipulated by tools and APIs that know the KDM metamodel.

REFERENCES

Aho, A. V., Sethi, R., and Ullman, J. D. (2007). *Compilers: principles, techniques, and tools*, volume 2. Addison-wesley Reading.

Almási, G. and Padua, D. (2002). Majic: Compiling matlab for speed and responsiveness. In *ACM SIGPLAN Notices*, volume 37, pages 294–303. ACM.

Anderson, J. C., Lehnardt, J., and Slater, N. (2010). *CouchDB: the definitive guide.* " O'Reilly Media, Inc.".

Bostock, M., Ogievetsky, V., and Heer, J. (2011). D^3 data-driven documents. *IEEE transactions on visualization and computer graphics*, 17(12):2301–2309.

Chen, X. (2010). Extraction and visualization of traceability relationships between documents and source code. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 505–510. ACM.

Doherty, J. and Hendren, L. (2012). Mcsaf: a static analysis framework for matlab. In *European Conference on Object-Oriented Programming*, pages 132–155. Springer.

Holten, D. (2006). Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE Transactions on visualization and computer graphics*, 12(5):741–748.

Joisha, P. G. and Banerjee, P. (2003). The magica type inference engine for matlab®. In *International Conference on Compiler Construction*, pages 121–125. Springer.

Lessa, I. M., Carneiro, G. d. F., Monteiro, M. P., and e Abreu, F. B. (2015a). On the use of a multiple view interactive environment for matlab and octave program comprehension. In *International Conference on Computational Science and Its Applications*, pages 640–654. Springer.

Lessa, I. M., de Figueiredo Carneiro, G., Monteiro, M. P., and e Abreu, F. B. (2015b). Scaffolding matlab and octave software comprehension through visualization. In *SEKE*, pages 290–293.

Li, Y. and Manoharan, S. (2013). A performance comparison of sql and nosql databases. In *Communications, Computers and Signal Processing (PACRIM), 2013 IEEE Pacific Rim Conference on*, pages 15–19. IEEE.

Monteiro, M., Cardoso, J., and Posea, S. (2010). Identification and characterization of crosscutting concerns in matlab systems. In *Conference on Compilers, Programming Languages, Related Technologies and Applications (CoRTA 2010), Braga, Portugal*, pages 9–10.

Munzner, T. (2014). *Visualization analysis and design*. CRC press.

OMG, O. M. G. (2016). Architecture-driven modernization: Knowledge discovery meta-model (kdm). *No Informado*.

Radpour, S., Hendren, L., and Schäfer, M. (2013). Refactoring matlab. In *International Conference on Compiler Construction*, pages 224–243. Springer.

Seriai, A., Benomar, O., Cerat, B., and Sahraoui, H. (2014). Validation of software visualization tools: A systematic mapping study. In *Software Visualization (VISSOFT), 2014 Second IEEE Working Conference on*, pages 60–69. IEEE.

Shneiderman, B. (1992). Tree visualization with tree-maps: 2-d space-filling approach. *ACM Transactions on graphics (TOG)*, 11(1):92–99.

Storey, M.-A. (2006). Theories, tools and research methods in program comprehension: past, present and future. *Software Quality Journal*, 14(3):187–208.