Towards a framework for idioms mining in model artefacts

The search for and need of a general framework for graph mining

Jens Van der Plas
Vrije Universiteit Brussel, Belgium
jevdplas@vub.be

Kelly Garcés
Universidad de los Andes, Colombia
kj.garces971@uniandes.edu.co

Coen De Roover
Vrije Universiteit Brussel, Belgium
cderoove@vub.be

Nicolás Cardozo
Universidad de los Andes, Colombia
n.cardozo@uniandes.edu.co

Abstract

Model-driven Engineering advocates the representation of any software artefact through a model. We want to support software engineers by finding frequently occurring idioms in Domain-specific programs from which a model can be extracted. To this end, we want to build a tool that can find such idioms. In order to build this tool, we need a framework for frequent subgraph mining. However, it seems that there are no adequate frameworks available. Towards the construction of this framework, we identify the main requirements this tooling must satisfy and propose a global architecture based on frequent-subgraph mining algorithms. Our claims and proposal are based on preliminary results of applying existing algorithms in ETL model artefacts.

1 Motivation

Model-driven engineering (MDE) can be used for fast implementation of Domain Specific Languages (DSL). MDE helps defining the concepts and relations of the domain by means of metamodel elements. In addition, MDE is able to create a model from the textual representation of a given program.

With the increased adoption of DSLs, it is important to supply software engineers with tools to facilitate and support the software development process. In this context, we want to create a tool to mine DSL programs to find new types of idioms or to recognise already known patterns in model artefacts. These idioms can then for example be used to identify *bad smells*, or as a basis for implementing new refactorings in current development tools.

2 Preliminary research

As preliminary research, we focused on finding idioms in programs written in the Epsilon Transformation Language (ETL), which is a DSL for model transformation specifications. We selected this DSL because of the availability of a relatively large dataset of artefacts, which is essential for any mining.

The tools for mining these particular kind of artefacts are scarce and mostly look for already existing bad smells [4]. In this context,

catalogues of bad smells for model transformations have been created [2, 8, 9, 10]. However, currently proposed tools do not allow finding new idioms. We do believe, that this issue can be extrapolated to other DSLs.

When reviewing the literature, we found that frequent subgraph mining algorithms can be helpful for finding idioms and we tried to apply some of them to ETL programs. We identified five steps to use these algorithms in model artefacts. First, the model artefacts have to be transformed into a structured representation, for example into an Abstract Syntax Tree (AST). Second, this representation has to be mapped onto a graph that can serve as an input for a frequent subgraph mining algorithm. We aim at using an existing framework or algorithm implementation. In a third step, we can use the mining algorithm to find frequently occurring patterns. In a fourth step, the output of the mining algorithm has to be mapped back onto the original input representation or onto another useful representation. Lastly, the results can be interpreted.

When using a frequent subgraph mining algorithm, there are certain parameters that can be varied in order to change the performance and tune the results of the algorithm. A first example of a parameter that can be varied is the mapping of the structured representation onto the graph. In addition, this type of algorithm requires a threshold parameter that indicates when a pattern can be treated as being frequent, which is a second parameter that can be varied. Lowering this threshold will find more patterns but will make the search slower.

The Epsilon 1.3 framework gives access to the AST representation of ETL files, which we will use to mine frequent idioms. We decided to look for existing tools and mining algorithms in order to use these to do the actual graph mining. For this, we looked at some existing frameworks which will be discussed in the next section.

2.1 Difficulties when using existing graph mining frameworks

2.1.1 GraMi

Followed approach

GraMi [3] is a script-based graph mining tool. Its input needs to be in a specific format, which is a file consisting of nodes with an id and a label, and vertices that connect two nodes. Each vertex also has a label. The node ids exclusively serve the purpose of providing a means for specifying the endpoints of the edges. All labels and ids are numbers. GraMi can interpret this graph as being directed or undirected. For our research, we will use directed graphs. In order to use this library, we have thus to map the AST of the model artefact to this representation. GraMi will use the graph structure and the labels to find frequent subgraphs.

We decide to map every node of the AST to a node in the graph and label these nodes with the type of the AST node. In order to know what these labels represent, we keep a mapping from the labels to the original text of the node, or to a special type (e.g. for variables which have the same AST node type but a different node text). Every edge in the graph will be labelled with the position of the child respective to its parent, e.g. the edge from a node corresponding to a parent and a node corresponding to a child will have label 1 if the child is the first child from that parent in the AST. This way, we can represent the order of the children in the AST in the graph. Another possibility would be to connect each pair of adjacent siblings from the AST in the graph. This method gives us the ability to use the edge labels freely, but might result in

finding too many patterns.

Following example illustrates this approach. Listing 1 shows an extract from an ETL file containing a pre-block and a transformation.

```
pre {
  "Inicio de transformacion M2M
     Business2Gui".println();
}
rule Business2Gui
transform b:business!Business
to g:gui!GUI{
  g.forms.add(b.getForm());
}
```

Listing 1: Extract from an ETL file

The AST of this ETL extract will be converted as described. An extract of the resulting file is shown in listing 2.

```
v 0 82
v 1 77 // A vertex with id 1 and
    label 77.
v 2 62
[...]
v 27 63
v 28 46
e 0 1 0 // An edge with label 0
   connecting vertices 0 and 1.
e 1 2 0
[...]
e 25 27 1
e 27 28 0
```

Listing 2: Extract from the graph file

Every node corresponds to a node in the AST from the ETL file. The labels correspond to the type of the AST node. AST nodes can for example represent values of a specific type (where there exists a label for each primitive type), an operator, type annotation and a variable. The node ids are used to create the edges.

As can be seen in listing 2, the input format required by GraMi only contains numbers, which makes the framework broadly applicable. However, this also means the mining algorithm only has a minimum of information, being the labels and graph structure. It is not possible to give more information to the algorithm. For example, we might want to label each node with a type and text, e.g. we want a node with a type "variable" and text "x". However, this is clearly not possible, which limits the applicability of GraMi for our problem.

```
=
 -> .
  -> _FEATURECALL_
  -> _FEATURECALL_
```

Listing 3: An example of a pattern. The arrows only serve to visualise the tree structure.

When applying this method to multiple ETL files, the resulting graph might consist of thousands of nodes. After mining these graphs for frequent subgraphs, the resulting graphs need to be converted back to tree representations. Listing 3 shows an example of a pattern, in which we just print the tree representation and the type or text of the AST nodes. The identified pattern represents an assignment from which the left-hand side has a specific form. As can be seen, this result is not very useful.

Encountered difficulties

Using this schema with GraMi, we encounter three major difficulties.

First, we notice that even executing the algorithm on a rather small graph and with a rather high threshold tends to be slow and requires a considerable amount of memory when executing this on a 2013 Macbook Pro with a 2,8 GHz Intel Core i7 processor and 16GB 1600 MHz DDR3 memory.

Second, the algorithm does not only output

maximal frequent subgraphs, but also all subgraphs of these. For this research, we are only interested in the maximal frequent subgraphs, since the others do not provide additional information. Moreover, these subgraphs clutter the results and make them more difficult to interpret.

Third, it is not possible to map the found frequent subgraphs to positions in the input in which they appear. Moreover, since we use generic labels for variables etc., the output does not tell us which variables are involved in the pattern. Also, in the current setup, these variable names may vary.

Possible variations

The approach we presented here can be altered and varied to change the results of the algorithm. For example, every node which symbolises a variable is now assigned the same label. This way, we only know that there should be *a* variable in the resulting patterns, but we do not know its name. In every occurrence of this pattern in the original files, this variable can be the same or not. Another possibility would be to assign every variable another label, depending on its name. In every discovered pattern, we then know the name of the variable. This will also restrict the amount of patterns that will be found. The same applies amongst others to operators and model names.

2.1.2 Arabesque

A second framework we looked at is Arabesque, which runs on Apache Spark [7]. However, we encountered some problems while setting installing the framework. As it turned out, due to a change in an API, Arabesque needs an older version of Apache Spark. This means, however, that using this implementation hampers the applicability of any approach based on it, since it could then only be used on clusters running the older version of Spark.

2.1.3 Other graph mining frameworks

For this research, we also looked at some other frameworks. When looking at Pegasus [1], we encountered the same limitation as with Arabesque.

2.2 Requirements for graph mining algorithms

From the difficulties we encountered, we derive three requirements a graph mining algorithm should have in order to be usable for our purpose:

- The implementation should have an acceptable performance.
- The implementation should only output maximal frequent subgraphs.
- It should be able to map the output of the algorithm to its input, i.e. not only should we be able to detect what frequent patterns occur, but we also need to know where they occur in the original DSL programs.

3 State of the art

In this section we discuss the state of the art. First, we look at research concerning finding patterns in model artefacts, which is also the topic of our research.

3.1 Mining model artefacts

Existing research concerning finding patterns in model artefacts mostly looked for already

known patterns, for example bad smells [2, 8, 9, 10]. Although this allowed researchers to find those bad smells in existing projects, it did not allow them to find new patterns, as they could only find patterns that were already catalogued. In this aspect, these previous studies differ from what we are doing.

Additionally, research to discover unknown patterns has also been conducted. This research focused on finding systematic edits to software repositories [5]. However, there are some key differences between this research and our intentions which implies that their approach cannot be used for our problem. The biggest difference lies in the use of a frequent itemset mining algorithm, whereas our study requires a frequent subgraph mining algorithm. Next to the use of different mining algorithms, we also note that in our case it is not possible to access children in an AST by means of *properties* as is done by [5], since the AST that can be obtained from a graph artefact by means of the Epsilon library does not contain these properties. However, the notion of *equivalence criteria* does apply to our problem. This criterion can be used in a flexible manner to alter the kind and amount of patterns the algorithm will find. For example, when finding frequent patterns in an ETL file, the question whether variable names are important or not is a matter of equivalence.

In [4], an algorithm is presented for pattern mining in models independent of their metamodels by means of QVT. The approach combines an algorithm for frequent itemset mining combined with a heuristic. However, only the algorithms are presented and no framework or implementation is made available. In [6] an approach for mining JavaScript code is presented. Their approach however is limited to mining JavaScript code.

3.2 Frequent subgraph mining

When looking at the existing literature concerning graph mining, we make a distinction between actual graph mining algorithms and their implementation. Existing research mostly focused on finding new and better algorithms, for a specific definition of *better*. In order to be able to use those results, we need to find an implementation of those algorithms. However, they seem to be mostly unavailable. Second, we discover that most research concerning graph mining algorithms dates back from the beginning of the millennium. This would imply that eventual implementations resulting from the respective research are probably using outdated libraries.

In sections 2.1.1 and 2.1.2 we already discussed GraMi and Arabesque. We also mentioned Pegasus. It turns out to be very difficult to find implementations of graph mining algorithms that fulfill our requirements as spelt out in section 2.2. Moreover, since some existing implementations are based on old libraries, it is sometimes not possible to test them. Also, documentation from these algorithms, for example regarding their output, tends to be sparse.

4 Desirable features of a graph mining algorithm

The requirements stated in section 2.2 and the state of the art as stated in section 3 have led us to decide that it is not feasible to use existing implementations, since they do not meet our requirements. Therefore, we have sketched the requirements and architecture of a graph-based framework for mining model artefacts, which consists of an implementation of mining algorithms, and a means of facilitating using that implementation for a variety of purposes. In this section we present an outline

of the requirements we want our implementation to have, in order to make it durable and broadly usable. By stating these requirements, we want to make sure the framework we will build will not suffer from the same shortcomings than those identified in section 2.

First of all, the framework should be generic so that it can be used to mine different types of model artefacts. It should be able to use the framework with different types of input and output. To be able to do so, the framework should use its own kind of internal graph representation, which can be built for different types of input files. For this, the grammar of the input file type could be used. For example, when analysing Java programs, a grammar for Java could be given to the framework. This approach guarantees our framework is generically applicable on different types of input.

The internal graph representation should contain useful information so that the mining can be customised. For example, it has to be possible to decide whether variable names need to match or not, which is a variation described in section 2.1.1. The same reasoning can be applied to functions, types, etc. This implies that the internal graph representation must have a means to differ nodes by e.g. a type. For each kind of file that should be mined, a mapping from the file text to the graph representation should be defined. The mapping can, but does not necessarily has to, be based on the AST of the file(s). This part of the framework can be seen as a language specific plug-in. Model artefacts could be defined in terms of a grammar or metamodel. This mapping can be seen as an instrument to alter the results found by the mining algorithm in order to tune the algorithm.

The framework's output should not only contain the frequent patterns that were found, but also where these patterns occur in the input. For each frequent subgraph, we should be able to find all instances of the subgraph in the input. A graph node has thus at least following three properties: (1) the text of the AST node in the input which it represents (a string), (2) a reference to the position of the AST node in the input and (3) a type indicator, which indicates what the AST node represents, e.g. a variable, function, type, ...

The framework should have an acceptable performance. For this we think it is best to build it on top of Apache Hadoop or Spark. However, the framework should also be as standalone as possible in order to avoid the troubles we had when trying to use other frameworks like Arabesque. When using other frameworks, an intermediate abstraction layer could be used to prevent big changes to our framework when an API changes. It will also allow us to change the other libraries and frameworks that are used without having to make a lot of changes to our framework itself.

These requirements will make the framework robust and generic, so it can be used for a long time for mining graphs in a variety of different DSLs. The requirements for the framework give rise to a modular, layered design. Figure 1 gives a graphical representation of this design.

The design consists of a fixed mining component that takes an internal graph and mines this graph. The mining component has a layered design to improve modularity and flexibility. As explained in the fifth paragraph of this section, it is recommended to use an intermediate functional layer to prevent troubles when an API changes. Without this intermediate layer, it is possible a newer version of a dependency from which the interface has changed can not be used without making significant changes to the framework itself.

A second component is language specific.

This component needs to provide functionality to parse input files and transform them to graphs. This is language specific and it is not important for the mining itself how this should happen. In this paper we specified that we used a transformation based on an AST, but this need not be the case. For example, the Epsilon 1.4 library does not allow retrieval of ASTs anymore. When mining files using this library, another representation needs to be used to create the graph representation, such a representation could be the model itself. This implementation logic is hidden from the mining logic. Changing this module allows mining files written in different languages.

5 Applicability

We aim at making our framework as generic as possible. We hope this will make it useful for a variety of DSLs. In this section we provide some examples of potential uses of a frequent graph mining algorithm that is specialised in finding idioms in code.

A first example is supporting model-driven engineers, as explained in section 1. The framework could be used to mine frequent idioms. Hence, these idioms can be used to build new tools to support MDE or to identify new bad practices. Also new design patterns can be discovered. The same applies for programs written in other languages, even when not in the field of MDE. Hence, our tool could support all software engineers.

However, since the input can be varied, our framework could also be applied for purposes distinct from source code or graph artefact mining. In fact, any input that can be transformed into our internal graph transformation can be mined. This implies that our framework would be broadly applicable for all types of DSLs. For each DSLs type, however, a transformation to the internal graph representation should be specified.

6 Conclusion and future work

In this paper, we described our vision of an approach for mining frequently occurring patterns in model artefacts. To this end, we want to use a framework for frequent subgraph mining which should meet some requirements. However, no adequate usable framework seems to be available. To solve this problem, we propose a framework for frequent subgraph mining that meets our requirements and describe some of the properties that the framework needs to have. We concluded by summing up some applications for which our framework could be used.

In order to build the proposed framework, further steps in our research include:

- Looking into graph mining algorithms to decide which is/are the most feasible to implement in our framework. This also includes verifying whether the algorithm allows us to comply with the requirements listed in section 2.2.
- Deciding which internal graph representation will be used in order to give as much information to the mining algorithm as possible. Also, the type of grammars to be used and how to use these grammars to create a graph needs to be investigated. Different approaches might be possible here. For models, a metamodel could for example be used instead of a grammar.
- Finally, after the decisions above have been taken, the framework has to be implemented while complying with the specifications as listed in section 4.

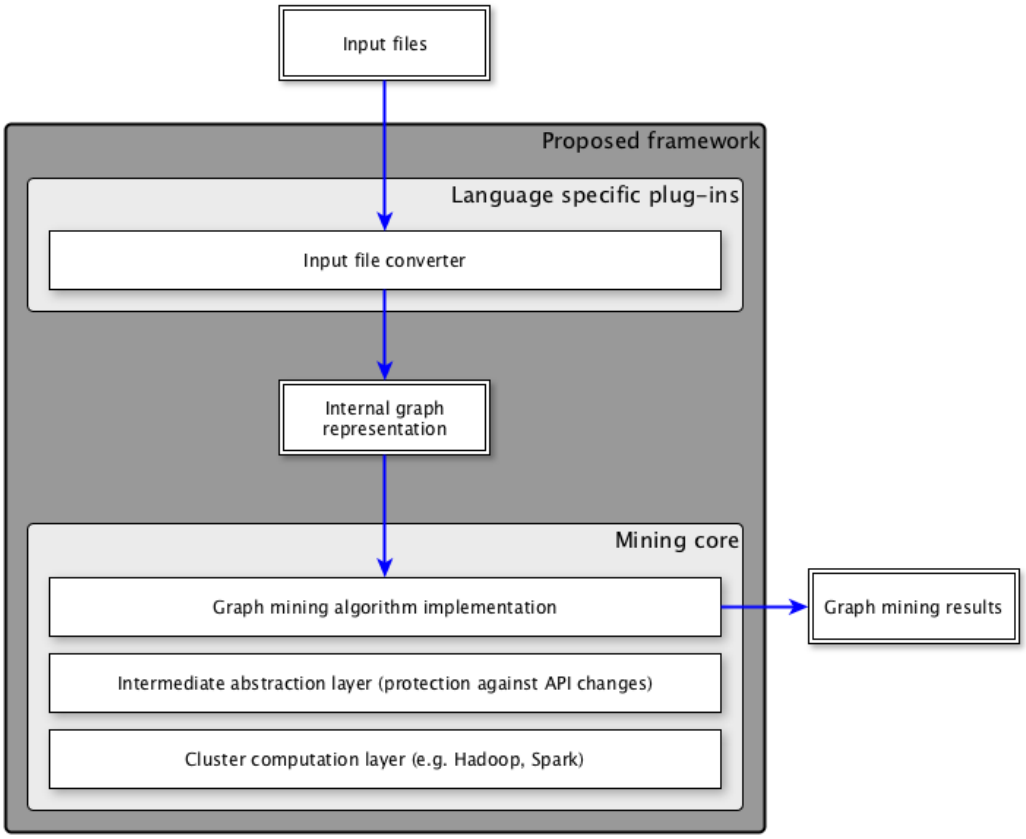

Figure 1: Design of the proposed framework

After these steps are performed, the mining framework can be used to find idioms in model artefacts. Relying on the unification power of models, our new framework will then allow us to mine many kinds of files easily.

References

[1] Pegasus an award-winning, open-source, graph-mining system with massive scalability. http://www.cs.cmu.edu/ pegasus/. Accessed: 2017-08-02.

[2] Nicolás Bonet, Kelly Garcés, Rubby Casallas, María E. Correal, and Ran Wei. Influence of programming style in transformation bad smells: Mining of etl repositories, 2017.

[3] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. Grami: Frequent subgraph and pattern mining in a single large graph. *Proc. VLDB Endow.*, 7(7):517–528, March 2014.

[4] Jens Kübler and Thomas Goldschmidt. A pattern mining approach using qvt. In *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications*, ECMDA-FA '09, pages 50–65, Berlin, Heidelberg, 2009. Springer-Verlag.

[5] Tim Molderez, Reinout Stevens, and Coen De Roover. Mining change histories for unknown systematic edits. In *Proceedings of the 14th International Conference on Mining Software Repositories*,

MSR '17, pages 248–256, Piscataway, NJ, USA, 2017. IEEE Press.

[6] Hung Viet Nguyen, Hoan Anh Nguyen, Anh Tuan Nguyen, and Tien N. Nguyen. Mining interprocedural, data-oriented usage patterns in javascript web applications. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 791–802, New York, NY, USA, 2014. ACM.

[7] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. Arabesque: A system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 425–440, New York, NY, USA, 2015. ACM.

[8] Matthias Tichy, Christian Krause, and Grischa Liebel. Detecting performance bad smells for henshin model transformations. In Benoit Baudry, Jürgen Dingel, Levi Lucio, and Hans Vangheluwe, editors, *Proceedings of the Second Workshop on the Analysis of Model Transformations (AMT 2013), Miami, FL, USA, September 29, 2013*, volume 1077 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2013.

[9] Ran Wei and Dimitris S. Kolovos. Automated analysis, validation and suboptimal code detection in model management programs. In Dimitris S. Kolovos, Davide Di Ruscio, Nicholas Drivalos Matragkas, Juan de Lara, István Ráth, and Massimo Tisi, editors, *Proceedings of the 2nd Workshop on Scalability in Model Driven Engineering co-located with the Software Technologies: Applications and Foundations Conference, BigMDE@STAF2014, York, UK, July 24, 2014.*, volume 1206 of *CEUR Workshop Proceedings*, pages 48–57. CEUR-WS.org, 2014.

[10] Manuel Wimmer, Salvador Martínez, Frédéric Jouault, and Jordi Cabot. A catalogue of refactorings for model-to-model transformations. *Journal of Object Technology*, 11(2):2:1–40, August 2012.