Fragment Retrieval on Models for Model Maintenance: Applying a Multi-Objective Perspective to an Industrial Case Study

Francisca Pérez*, Raúl Lapeña, Jaime Font, Carlos Cetina

Universidad San Jorge. SVIT Research Group
Autovía A-23 Zaragoza-Huesca Km.299, 50830, Zaragoza, Spain

Abstract

Context: Traceability Links Recovery (TLR), Bug Localization (BL), and Feature Location (FL) are amongst the most relevant tasks performed during software maintenance. However, most research in the field targets code, while models have not received enough attention yet.

Objective: This paper presents our approach (FROM, Fragment Retrieval on Models) that uses an Evolutionary Algorithm to retrieve the most relevant model fragments for three different types of input queries: natural language requirements for TLR, bug descriptions for BL, and feature descriptions for FL.

Method: FROM uses an Evolutionary Algorithm that generates model fragments through genetic operations, and assesses the relevance of each model fragment with regard to the provided query through a fitness configuration. We analyze the influence that four fitness configurations have over the results of FROM, combining three objectives: Similitude, Understandability, and Timing. To analyze this, we use a real-world case study from our industrial partner, which is a worldwide leader in train manufacturing. We record the results in terms of recall, precision, and F-measure. Moreover, results are compared against those obtained by a baseline, and a statistical analysis is performed to provide evidences of the significance of the results.

Results: The results show that FROM can be applied in our industrial case study. Also, the results show that the configurations and the baseline have significant differences in performance for TLR, BL, and FL tasks. Moreover, our results show that there is no single configuration that is powerful enough to obtain the best results in all tasks.

Conclusions: The type of task performed (TLR, BL, and FL) during the retrieval of model fragments has an actual impact on the results of the configurations of the Evolutionary Algorithm. Our findings suggest which configuration offers better results as well as the objectives that do not contribute to improve the results.

Keywords: Conceptual Models, Traceability Links Recovery, Bug Localization, Feature Location, Evolutionary Algorithms

1. Introduction

Amongst the most common and relevant tasks in the Software Engineering field, especially when maintaining software products, are Traceability Links Recovery, Bug Localization, and Feature Location [1, 2, 3, 4]. To tackle these tasks, Information Retrieval

*Corresponding author. Tel.: +34 976060100
Email addresses: mfperez@usj.es (Francisca Pérez), rlapena@usj.es (Raúl Lapeña), jfont@usj.es (Jaime Font), ccetina@usj.es (Carlos Cetina)

(IR) techniques, such as Latent Semantic Indexing (LSI) [5, 6], have been used successfully [7, 8]. However, most research targets code [4, 3, 9], neglecting other software artifacts such as models. Models raise the abstraction level using concepts that are much less bound to the underlying implementation and technology and are much closer to the problem domain [10]. The practice of Model Driven Engineering has proved to increase efficiency and effectiveness in software development [10].

To increase the automation level when Traceability Links Recovery, Bug Localization and Feature Location are performed over models, we propose an approach named Fragment Retrieval on Models (*FROM*). Our approach uses a Multi-Objective Evolutionary Algorithm to retrieve the most relevant model fragments for different types of queries (natural language requirements for Traceability Links Recovery, bug descriptions for Bug Localization, and feature descriptions for Feature Location). To guide the Evolutionary Algorithm, we use three fitness objectives: Model Similitude through Latent Semantic Indexing (LSI) [5, 6], Model Understandability through Model Size [11, 12], and Model Timing through the Defect Principle [13, 14].

Moreover, we combine the three objectives into a total of four configurations: (1) Similitude, (2) Similitude + Understandability, (3) Similitude + Timing, and (4) Similitude + Understandability + Timing. We analyze the impact of each configuration on the results of the Evolutionary Algorithm for Traceability Links Recovery, Bug Localization, and Feature Location. In order to carry out this analysis, we use the models, natural language requirements, bug descriptions, and feature descriptions, all of them from a real-world case study provided by our industrial partner, Construcciones y Auxiliar de Ferrocarriles (CAF)[1], which is a worldwide leader in train manufacturing.

We record the results of the Evolutionary Algorithm for each configuration and the baseline for each type of query in terms of recall, precision, and F-measure. Also, results are compared against those obtained by a baseline in order to put FROM in perspective of previous works. The baseline retrieves model fragments using model comparisons among models instead of using an evolutionary algorithm or LSI. Our findings reveal that there is not a unique configuration of objectives that retrieves the best results for all of queries. In other words, the usage of different fitness objectives configurations is required to optimize the results of the Evolutionary Algorithm for either Traceability Links Recovery, Bug Localization, or Feature Location. In addition, we provide evidences of the significance of the results by means of statistical analysis.

The rest of the paper is structured as follows: Section 2 presents a motivating example. Section 3 presents our approach. Section 4 describes the evaluation, the results, and the statistical analysis. Section 5 discusses the results. Section 6 presents the threats to validity. Section 7 reviews the related work. Finally, Section 8 concludes the paper.

2. Motivating Example

Despite Model-Driven Development has not had the expected widespread success so far, major players in the software engineering field (i.e., tool vendors, researchers, and enterprise software developers) foresee a broad adoption of model-driven techniques because of scenarios that demand more abstract approaches than mere coding [10]. Fostering modeling efforts brings benefits in industrial contexts in order to improve productivity, while ensuring quality and performance [10].

In a model-driven industrial context, companies tend to have a myriad of products with large and complex models behind. The models are created and maintained over long periods of time by different software engineers, and the engineers in charge of the maintenance tasks (Traceability Links Recovery, Bug Localization, and Feature Location) often lack knowledge over the entirety of the product details. Under these conditions, maintenance tasks consume high amounts of time and effort, without guaranteeing good results. Our industrial partner reported performing the maintenance tasks manually at least 25

[1] `www.caf.net/en`

times per week, costing them a total monthly amount of working time ranging from 43.3 to 66.7 hours.

Figure 1 depicts a model example, taken from a real-world train, specified using the Domain Specific Language (DSL) that formalizes the train control and management of the products manufactured by our industrial partner. The DSL has the expressiveness required to describe both the interaction between the main pieces of installed equipment, and the non-functional aspects related to regulation. It will be used through the rest of the paper to present a running example. For the sake of understandability and legibility, and due to intellectual property rights concerns, we present an equipment-focused simplified subset of the DSL.

Specifically, the example of the figure presents a converter assistance scenario where two pantographs (High Voltage Equipment) collect energy from the overhead wires, and send it to their respective circuit breakers (Contactors), which in turn send it to their independent Voltage Converters. The converters then power their assigned Consumer Equipment: the HVAC on the left (air conditioning system), and the PA (public address system) and CCTV (television system) on the right.

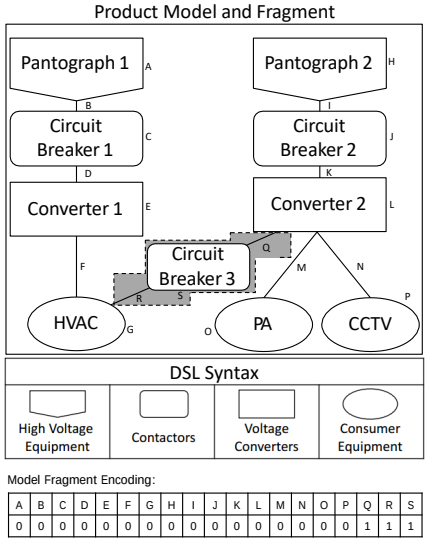


Figure 1: Example of model and model fragment

The elements of Figure 1 highlighted in gray conform an example model fragment, including one circuit breaker that connects Converter 2 to a Consumer Equipment assigned to Converter 1. This model fragment is the realization of the 'converter assistance' feature, which allows the passing of current from one converter to equipment assigned to its peer for coverage in case of overload or failure of the first converter.

A model fragment (which always belongs to a parent model) is encoded as a string of binary values that contains as many positions as elements in the parent model, where each position in the string has two possible values: 0 in case the element does not appear in the fragment, or 1 in case the element does appear in the fragment. In Figure 1, elements Q, R, and S conform the model fragment, so the corresponding values are set to '1' in its binary string representation.

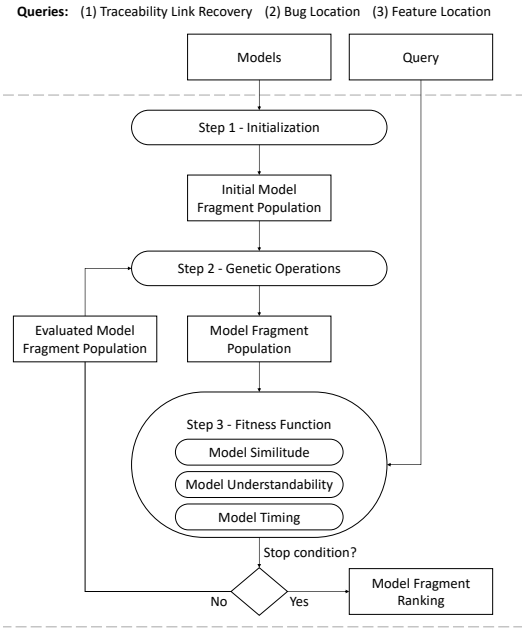
Although it may appear easy to locate the 'converter assistance' feature in the model, it becomes very complex in the models of our industrial partner where each train unit is specified through several thousand elements. According to our industrial partner, software engineers who belonged to the original team of modelers and who work on a monthly basis with the product involved in the example, are able to locate the feature in around 26 minutes. Another engineer, not related to the project but with knowledge of the products in the company, spent 34 minutes on the same task. Finally, two newcomer modelers spent around 40 minutes of combined work until they fulfilled the task, but they did so in a non-accurate manner. Considering these numbers, an approach that automatically retrieves model fragments is strongly needed.

3. Approach

The goal of the presented approach, *FROM* (Fragment Retrieval On Models), is to use an Evolutionary Algorithm to retrieve model fragments for Traceability Links Recovery, Bug Localization, and Feature Location. In addition, we use different combinations of fitness objectives as fitness function for the Evolutionary Algorithm in *FROM* in order to establish

which combination of objectives guides *FROM* to the best results.

Figure 2 shows an overview of *FROM*. The top part of the figure highlights the inputs (the three possible types of Natural Language (NL) queries and the models where the model fragment must be retrieved), the middle part shows the steps of the Evolutionary Algorithm (including the Fitness Objectives within the Fitness Function), and the bottom part presents the four Fitness Objective Configurations considered through this work.



	Config. 1	Config. 2	Config. 3	Config. 4
Similitude	✓	✓	✓	✓
Understandability	×	✓	×	✓
Timing	×	×	✓	✓

Figure 2: Overview of the approach and configurations

3.1. ***Queries***

In *FROM*, we consider Traceability Links Recovery, Bug Localization, and Feature Location.

1) Traceability Links Recovery: The functionality of software products refers to what tasks a product should be able to carry out, and also to how those tasks should be carried out. It is described through specifications, which usually take the shape of NL requirements documents. The objective of Traceability Links Recovery among requirements and models is to establish the model fragment that implements a particular NL requirement. For Traceability Links Recovery, the input query is a NL requirement, for which the model fragment must be retrieved. For example, a functional requirement of our case study is: 'The PLC will inhibit the connection with the pantograph whenever the lowering button in the active cabin is pushed, as long as the pantograph is in closed state and more than five seconds have passed after the closing of the circuit breaker, being the doors off'.

2) Bug Localization: When errors manifest in the expected functionality of a product, software engineers create bug report documents or incidence tickets, which adopt the form of NL descriptions of the errors. The objective of Bug Localization is to identify the model fragment that causes a particular error in a product model in order to fix it. For Bug Localization, the input query is one of the error descriptions. For example, a bug description taken from our case study is: 'In case of failure of the second converter, the third converter is expected to deviate 50% of its power to the HVAC within 2 seconds, but the converter assistance scenario does not activate'.

3) Feature Location: The term 'feature' refers to a particular characteristic that a product may include. The presence or absence of a characteristic in a product, in that sense, entails the existence of different product configurations. Feature Location is concerned with identifying software artifacts (in our case, model fragments) associated with such specific characteristics. For Feature Location, the input query is the description of a feature in NL. For example, a feature taken from our case study is: 'Enabled Cabin Detection: the system will automatically determine the cabin in use through the presence of a key in the control desk, and will automatically set it as the Enabled Cabin from which the train will be controlled'.

3.2. ***Evolutionary Algorithm***

Our approach relies on a Multi-Objective Evolutionary Algorithm (MOEA) [15] that iterates over

model fragments, modifying them using genetic operations. In a previous work [16], domain experts were requested to limit the search space by choosing a subset of the models, or by providing restrictions of elements that do not have to appear in the solutions. However, the search space was still very large (a model of 500 elements can yield around 10^{29} potential fragments). Evolutionary algorithms have obtained good results by addressing similar problems with large search spaces, so we have chosen to use an evolutionary algorithm. The output of the approach is a model fragment ranking for the input Traceability Links Recovery, Bug Localization, or Feature Location query. The MOEA runs in three steps:

1) Initialization: The first step of our approach is to, from the product models, generate a population of model fragments that serves as input for the genetic algorithm. In order to generate the population of model fragments, parts of the models are extracted randomly and added to a collection of model fragments.

In order to generate a random model fragment, we designed algorithm 1 (see Appendix A). The algorithm first selects a random initial model element E. Then, using E, a new model fragment F is created. In addition, a second element N, neighbor to E, is taken. A valid neighbor N is an element that is directly connected to E. In case there is more than one possible neighbor element, one of the possible neighbors is randomly chosen. Then, a random number of iterations are performed.

Notice that, due to the neighbor selection process, the algorithm returns a model fragment built with a subset of elements from the parent model which are contiguously connected. This algorithm only produces fragments that are part of the original model, it does not create new elements, and the resulting model fragments keeps the conformance to the metamodel.

2) Genetic Operations: The second step of our approach is to generate a set of model fragments that could realize the provided Traceability Links Recovery, Bug Localization, or Feature Location query. The generation of new model fragments, based on existing ones, is done by applying a set of two genetic operators adapted to work over model fragments: crossover, and mutation. Both are further described in the following paragraphs.

The **crossover operation** [17] enables the creation of a new individual by combining the genetic material from two parent model fragments. The crossover operation takes the model fragment from the first parent and the model from the second parent, and generates a new individual that contains elements from both parents through model comparisons. If the comparison finds the first model fragment in the second model, the operation creates a new individual with the model fragment taken from the first parent but referencing the product model from the second parent. Otherwise, the crossover returns the first parent unchanged. This operation broadens the search space to different models. The model fragments from the first parent and the new individual will be the same but, since parent and child can reference different models, they will often mutate differently and provide different individuals in further generations. The algorithm of this operation is outlined in Algorithm 2 (available in Appendix A).

The upper part of Figure 3 shows an example of the application of the crossover operation. First, we select the two parents to which the operator is applied. Then, the model fragment inside the first parent is compared with the second parent. Since the comparison is able to find the model fragment in the second parent, the process creates a new individual with the model fragment, referencing the second parent.

The **mutation operator** [18] is used to imitate the mutations that randomly occur in nature when new individuals are born. That is, new individuals hold small differences with their parents that could make them adapt better (or worse) to their living environment. Following this idea, the mutation operator applied to model fragments takes as input a model fragment and mutates it into a new one, which is returned as output. As the approach is looking for fragments of a product model that realize a particular feature, the new modified fragment must remain a part of the product model. Therefore, the modifications that can be done to the model fragment must be driven by the product model.

In particular, the mutation operator can perform two distinct modifications: addition of elements to

the model fragment, or removal of elements from the model fragment. To that extent, one of the two operations is firstly chosen. If the operation 'addition' is chosen (see the bottom-left part of Figure 3), an element of the fragment with connections to model elements that are not included in the fragment is chosen in order to add one of these non-included model elements to the fragment. For instance, the *Converter 1* element of the fragment has a connection with the *HVAC* model element, not included in the original model fragment. As result of the addition operation, a modified model fragment that includes the *HVAC* model element is obtained. If the operation 'removal' is chosen (see the bottom-right part of Figure 3), an element of the model fragment that is connected with only one other element of the model fragment is chosen to be removed from the model fragment. For instance, a modified model fragment that does not include the Pantograph 1 model element is obtained as a result of the removal operation. The algorithm of this operation is outlined in Algorithm 3 (available in Appendix A).

3) Fitness Function: The third step of the approach assesses each of the produced candidate model fragments, ranking them according to a fitness function. Our approach presents fitness functions based on combinations of three distinct fitness objectives, detailed in the following section.

3.3. ***Fitness Objectives***

In this section, details are provided for each objective.

1) Model Fragment Similitude: To assess the relevance of each model fragment with relation to the provided query, we apply methods based on Information Retrieval (IR) techniques. In particular, we apply Latent Semantic Indexing (LSI) [5, 6] to analyze the relationships between the model fragments in the population and the query.

However, results retrieved by LSI depend greatly on the style on which the Natural Language (NL) of the input is written. It is often regarded as beneficial to preprocess the inputs of LSI through Natural Language Processing (NLP) techniques [19] to improve LSI results. A frequent practice to achieve said preprocessing is to use a combination of Parts-of-Speech (POS) tagging, removal of stopwords, and stemming, as presented in [20].

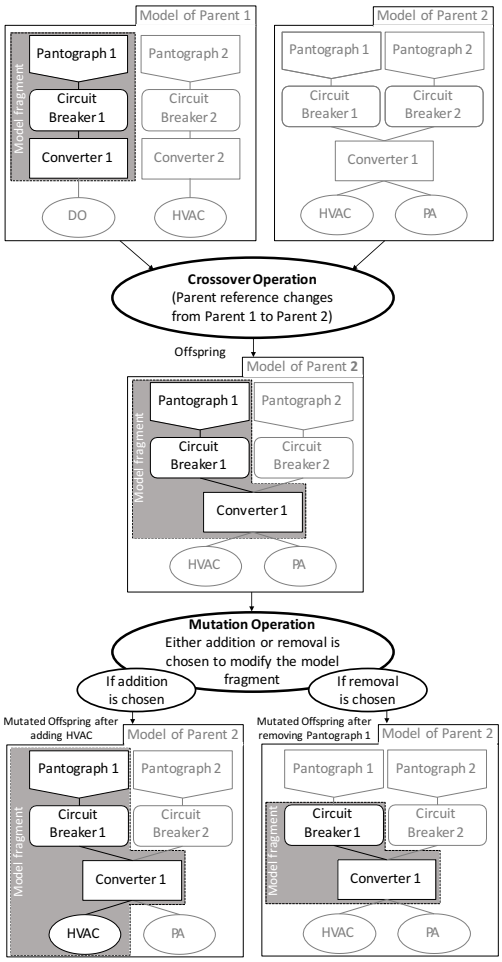


Figure 3: Genetic operations example

In our approach, we adopt said practice to process the NL from the model fragments and the queries. The NL texts of both the model fragments and the queries are preprocessed through the following steps: (1) our approach searches for domain terms provided by the software engineers in the text, saving them; (2) POS tagging is applied to the text, and the POS tags of the words are analyzed and filtered by their syntactic role, keeping only the nouns, as suggested by [20]; and (3) the remaining POS Tags are stemmed, and

refined with a set of stopwords, also provided by the software engineers. The stemmed POS Tags from Step 3 plus the saved domain terms from Step 1 build the text of the processed elements used as input for LSI.

Once the NL texts from both the model fragments and the query are processed, it is possible to apply the LSI technique. LSI constructs vector representations of a query and a corpus of text documents by encoding them as a term-by-document co-occurrence matrix. That is, a matrix where each row corresponds to *terms* and each column corresponds to *documents*, followed by the *query* in the last column. Each cell of the matrix holds the number of occurrences of a *term* inside a *document* or the *query*. In our approach, *terms* are all the individual words from the processed NL of model fragments and the query, the *documents* are the NL representations of model fragments, and the *query* is the provided Traceability Links Recovery, Bug Localization, or Feature Location query. To generate the *documents*, the model fragments are processed to extract the terms that correspond to the elements that conforms them. The words obtained this way for a particular model fragment conform its corresponding *document*.

Once the matrix is built, it is normalized and decomposed into a set of vectors using a matrix factorization technique called Singular Value Decomposition (SVD) [5]. SVD is a form of factor analysis, or more properly the mathematical generalization of which factor analysis is a special case. In SVD, a rectangular matrix is decomposed into the product of three other matrices. One component matrix describes the original row entities as vectors of derived orthogonal factor values, another describes the original column entities in the same way, and the third is a diagonal matrix containing scaling values such that when the three components are matrix-multiplied, the original matrix is reconstructed. In SVD, a 'k' value of dimensions is chosen as a tuning parameter, reducing the matrices accordingly. According to recent research, keeping a 'k' value of around 300 (or the maximum, if there are less than 300 dimensions) will usually provide the best possible results with moderate-sized document collections [21]. However, as stated by [22] and [23], the 'k' value should be studied and tuned for each approach individually in order to optimize the results. In their work, Khatiwada et al. [24] determine the 'k' parameter through a brute force strategy, generating several 'k' values and evaluating the performance of each of their datasets for every 'k' value. The tuning of the 'k' parameter for our work, however, is out of the scope of this paper, and as such we acknowledge it as future work.

Using SVD, one vector that represents the latent semantics of the NL texts is obtained for each *document* and for the *query*. Finally, the similarities between each *document* and the *query* are calculated as the cosine between both of their vectors, obtaining values between -1 and 1.

The top part of Figure 4 shows an example of co-occurrence matrix, taken from our approach (for space reasons, columns and rows are shown in a compact way). Each *document* column is a NL representation of one of the model fragments in the population. The *query* column is the provided input Traceability Links Recovery, Bug Localization, or Feature Location query. Each *term* row is one of the words extracted from the NL texts of model fragments and the provided Traceability Links Recovery, Bug Localization, or Feature Location query. Each cell shows the number of occurrences of each of the *terms* in the model fragments. The bottom left part of Figure 4 shows the result of applying the SVD technique to the matrix. The vector labeled with 'Q' represents the *query*, while the ones labeled as 'MF' represent *document* model fragments. Bottom right part of Figure 4 shows the scores of each model fragment, calculated by computing the cosine between their associated vector and the *query* vector.

2) Model Fragment Understandability. There are several metrics that measure different factors in models, such as their underlying complexity or their understandability by humans [11]. The findings published in [12] prove that there is a strong correlation between the size of a particular model and its understandability by a human modeler, therefore impacting the performance of the modeler when working with it. When presented with several models for an industrial solution, smaller models always entail better modeler performance results.

To measure the size of a model fragment, three

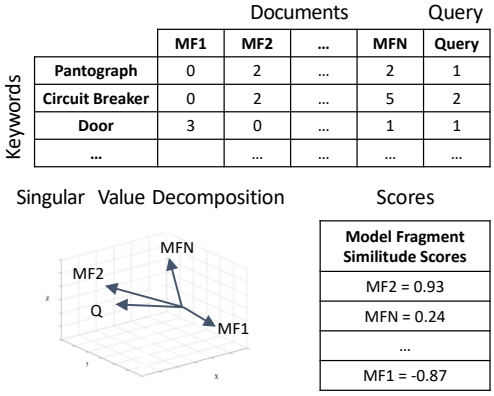


Figure 4: LSI example

metrics are defined in [12] and described in Appendix B: **U1**) Counting the number of elements in a model, **U2**) Weight factors per model element, and **U3**) Weight factors per model element per diagram type.

In [12], the author analyzes the results of applying the three metrics to a set of models, finding that the three metrics are extremely correlated, with none yielding significantly better results over the other two. Since it is easier to implement and compute, it is strongly suggested to use **U1**. Therefore, we use **U1** as the metric of choice for our Understandability Fitness Objective.

3) Model Fragment Timing. The Defect Principle, or Defect Localization Principle, states that the most recent modifications to a project are the most relevant for certain Information Retrieval purposes [25, 13, 14]. Through the Defect Principle, modification timespans can be considered and introduced as a Fitness Objective for Traceability Links Recovery, Bug Localization, and Feature Location.

Through this work, we carry out Traceability Links Recovery, Bug Localization, and Feature Location on models. Therefore, our aim is to retrieve the most relevant model fragments for a particular Traceability Links Recovery, Bug Localization, or Feature Location query. Model fragments are formed by model elements, and each model element has an associated modification time. When we apply the Defect Principle to model fragments, we have to decide how to assign a modification time to the model fragment from the modification time information on its model elements. There are four possible measurements of the modification timespans for the Defect Principle:

(1) **Most recent model modifications:** this measurement captures the modification timespan of the most recently modified model element.

(2) **Oldest model modifications:** this measurement captures the modification timespan of the least recently modified model element.

(3) **The mean of the modification timespan of the modified model elements:** the value of the measurement is the mean value of the modification timespans of the model elements.

(4) **The sum of the modification timespan of the modified model elements:** the value of the measurement is the sum of the modification timespans of the model elements.

Software engineers from our industrial partner, when faced with different model fragments, declared that those they had modified more recently were more familiar to them, thus easier to understand and work with. Therefore, we chose measurement (1) as the way to evaluate our Model Timing Fitness Objective.

The time difference is based on the number of days and can therefore be very large when the model fragment was modified a long time ago. To normalize the time difference, mathematical solutions such as square root or logarithm can be used. We used square roots because it has achieved good results in other works that use time differences [13].

Figure 5 shows an example of timespan for each model element of the model fragment highlighted in gray. For example, the *Circuit Breaker 1* has been modified 89 days ago. Since the most recent model modification is 17 days (from the *Convert1* model element), the value of the model fragment is 17 days that means a square root of 4.123.

3.4. Fitness Objectives Configurations

To compare how the distinct objective configurations affect *FROM* for the different types of queries, we have designed a total four possible objective configurations (shown in the bottom part of Figure 2):

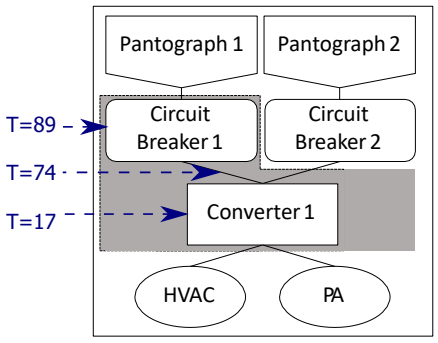


Figure 5: Timespan of the modifications of the model elements of a fragment

C1: Similitude.

C2: Similitude + Understandability.

C3: Similitude + Timing.

C4: Similitude + Understandability + Timing.

It is worth mentioning that creating configurations without the Similitude measurement is possible, but meaningless. Such configurations would produce the smallest and/or most recently modified model fragments in the case study, regardless on whether they had anything to do with the introduced query, rendering them useless.

4. Evaluation

This section presents the evaluation of our approach.

4.1. ***Research Questions***

There are several aspects that we want to evaluate with regard to how the different configurations affect FROM for the different types of queries. In order to address the evaluation of these aspects, we formulated the following research questions:

RQ$_1$: *How does the performance of the different objective configurations compare to the performance of the baseline for different query types?*

RQ$_2$: *Is the difference in performance between the objective configurations and the baseline significant?*

RQ$_3$: *Does the type of query have an impact on the performance of the different objective configurations?*

4.2. ***Baseline***

In order to put the performance of *FROM* in perspective and to relate our work to previous works, we compare it to a baseline for fragment retrieval in models. Traditionally, fragment retrieval in models has been performed through model comparisons among models [26, 27, 28, 29, 30, 31]. These works classify the elements based on their similarity and identify the dissimilar elements as the model fragments. The predominant technology of choice to implement their approaches is EMF Model Compare, which relies on Model Matching to perform the comparisons. Hence, the baseline is our implementation of the algorithms to retrieve fragments presented in [28], which also uses EMF Model Compare to perform the model comparisons as the previous works.

4.3. ***Experimental Setup***

FROM and the baseline are executed taking as input the query and the models provided by our industrial partner. Our industrial partner provided us with: 103 natural language requirements, 121 feature descriptions and 42 bug descriptions of their railway solutions. The models of 23 trains are specified through an average of 8250 model elements.

We executed 30 independent runs for each query and approach (the four configurations of *FROM* and the baseline) for *FROM* (as suggested by [32]), i.e., 103 (natural language requirements) x 5 (approaches) x 30 repetitions + 121 (feature descriptions) x 5 (approaches) x 30 repetitions + 42 (bug descriptions) x 5 (approaches) x 30 repetitions = 39900 independent runs.

Once the four configurations of *FROM* and the baseline are executed, we obtain as result a ranking of model fragments. Next, we take the best solution of the ranking (the model fragment at position 1) to compare it with an oracle, which is the ground truth. Once the comparison is performed, a confusion matrix is calculated.

A confusion matrix is a table often used to describe the performance of a classification model on a set of test data (the best solutions) for which the true values are known (from the oracle). In our case, each solution obtained is a model fragment composed of a subset of the model elements that are part of the product model. Since the granularity is at the level of model elements, each model element presence or absence is considered as a classification. The confusion matrix distinguishes between the predicted values and the real values classifying them into four categories:

- True Positive (TP): values that are predicted as true (in the solution) and are true in the real scenario (the oracle).
- False Positive (FP): values that are predicted as true (in the solution) but are false in the real scenario (the oracle).
- True Negative (TN): values that are predicted as false (in the solution) and are false in the real scenario (the oracle).
- False Negative (FN): values that are predicted as false (in the solution) but are true in the real scenario (the oracle).

Then, some performance measurements are derived from the values in the confusion matrix. In particular, we create a report including three performance measurements: recall, precision, and F-measure for the baseline and configurations for each type of query.

Recall measures the number of elements of the solution that are correctly retrieved by the proposed solution and is defined as follows:

$$Recall = \frac{TP}{TP + FN}$$

Precision measures the number of elements from the solution that are correct according to the ground truth (the oracle) and is defined as follows:

$$Precision = \frac{TP}{TP + FP}$$

F-measure corresponds to the harmonic mean of precision and recall and is defined as follows:

$$F - measure = 2 * \frac{Precision * Recall}{Precision + Recall}$$

Recall values can range between 0% (no single model element obtained from the oracle is present in any of the model fragments of the solution) to 100% (all the model elements from the oracle are present in the solution). Precision values can range between 0% (no single model fragment from the solution is present in the oracle) to 100% (all the model fragments from the solution are present in the oracle). A value of 100% precision and 100% recall implies that both the solution and the oracle are the same.

At this point, it is important to highlight that the fitness objective Configuration 1 (Similitude) is a Single-Objective Evolutionary Algorithm (SOEA), whereas the other three configurations are MOEA. For this reason, other common MOEA measures such as hypervolume [33] are not necessarily suitable for comparing solutions by MOEAs with solutions by SOEAs as the work in [34] shows.

Therefore, in order to compare the results, we first take the best solution of Configuration 1 for its single-objective (similitude with the query). Second, we take the best solution of each of the other configurations with regard to the objective of Configuration 1 (similitude with the query) as described in [34].

4.4. ***Oracle Preparation***

The oracle was provided by our industrial partner, since the model fragments that realize each of the 103 requirements, 121 features and 42 bugs were already documented. It is also worth noting that the oracle has not been created for this evaluation, and that many of the provided model fragments were created by engineers who are currently not working in the company. We checked both that there were no queries without model fragments, and that the model fragments were in the models provided for the evaluation.

4.5. *Implementation Details*

FROM[2] is based on NSGA-II [35], one of the most frequently used Multi-Objective Evolutionary Algorithms. Given a population of model fragments where each model fragment has up to three fitness values (see Subsection 3.4), NSGA-II orders these model fragments by means of non-dominated sorting. A model fragment is non-dominated when there is no other model fragment that improves any fitness value without worsening other fitness value. As a result, NSGA-II finds pareto-optimal model fragments.

The rest of the settings such as population size, crossover probability, and mutation probability are detailed in Table 1. For those settings, we have chosen values that are commonly used in the literature [36]. The values are 100, 0.9, and 0.1, respectively.

Table 1: Parameter settings

Parameter description	Value
$Size$: Population Size	100
μ: Number of Parents	2
λ: Number of offspring from μ parents	2
r: Solutions replaced at population size	2
$p_{crossover}$: Crossover probability	0.9
$p_{mutation}$: Mutation probability	0.1

In general, there are two atomic performance measures for evolutionary algorithms: one regarding solution quality and one regarding algorithm speed or search effort. In this paper, we focus on the solution quality (i.e., obtaining a solution that is more similar to the one from the oracle in terms of precision and recall). After running some prior tests for each fitness configuration to determine the time to converge (and adding a margin to ensure convergence), we allocated a fixed amount of wall clock time (80 seconds) to stop the execution. During that time, our algorithm is capable of executing an average of 7307 generations (with an standard deviation of 1500 generations). We performed the execution of FROM using an array of computers with processors ranging from 4 to 8 cores, clock speeds between 2.2 GHz and 4GHz, and 4-16 GB of RAM. All of them were running Windows 10 Pro N 64 bits as the hosting Operative System and the Java(TM) SE Runtime Environment (build 1.8.0 73-b02).

We have used the Eclipse Modeling Framework to manipulate the models and the Common Variability Language (CVL) [37] to manage the model fragments. The NLP techniques used to process the language have been implemented using OpenNLP [38] for the POS Tagger (accounting for an 88% precision [39]) and the English (Porter 2) [40] stemming algorithm. LSI has been implemented using the Efficient Java Matrix Library [41]. The genetic operations are built upon the Watchmaker Framework for Evolutionary Computation [42].

The available implementation presented in FROM is limited by confidentiality agreements in force with our industrial partner, since the approach is currently in use, and the trains of the case study are currently operating and under maintenance contracts.

4.6. *Research Question 1*

To answer how is the performance of the configurations and the baseline, this subsection presents the results of performance. Figure 6 shows the charts with the recall and precision results for the configurations and the baseline (rows of the figure) and the type of query (columns in the figure). A dot in the graphs represents the average result of recall and precision for the 30 repetitions.

RQ_1 answer. Table 2 shows the mean values of recall, precision and F-measure of the graphs for the four configurations and the baseline (rows) in Traceability Links Recovery, Feature Location and Bug Localization (columns). In Traceability Links Recovery, Configuration 1 obtains the best results in recall and precision, providing and average value of 54.33% in recall and 59.93% in precision. In Feature Location, Configuration 2 obtains the best results in recall and precision, providing and average value of 73.29% in recall and 70.60% in precision. In Bug Localization, Configuration 4 obtains the best results in recall and precision, providing and average value of 84.91% in recall and 79.94% in precision.

[2] `https://bitbucket.org/svitusj/from`

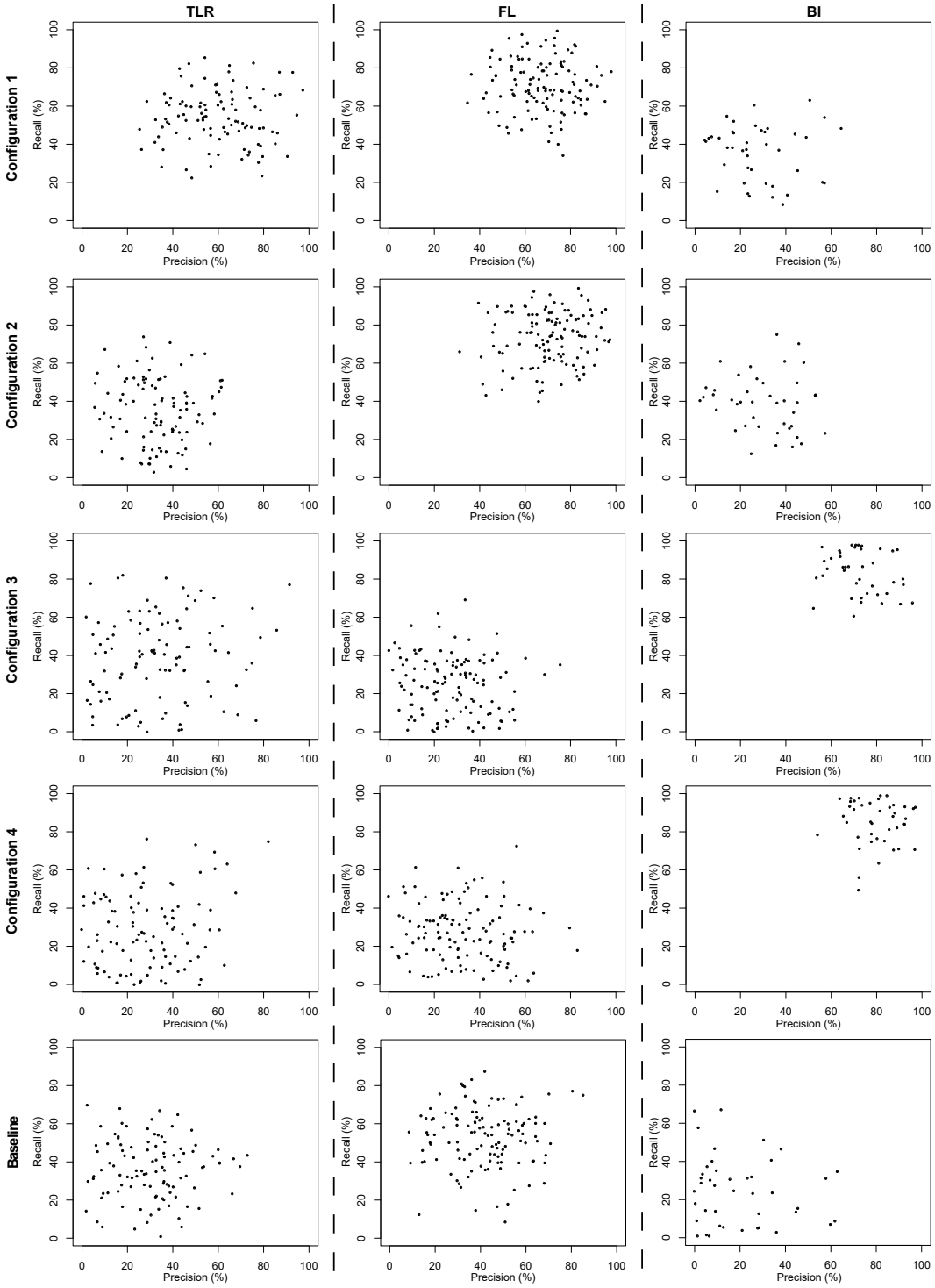


Figure 6: Mean Recall and Precision values for *FROM* and the baseline

Table 2: Mean Values and Standard Deviations for Recall, Precision and F-Measure

	Recall ± (σ)		
	TLR	FL	BL
Configuration 1	54.33±14.23	70.95±13.59	35.95±14.49
Configuration 2	35.21±17.05	73.29±13.65	39.47±14.73
Configuration 3	38.72±22.14	25.04±15.06	83.09±11.28
Configuration 4	29.07±19.66	27.85±15.05	84.91±11.85
Baseline	36.14±15.55	58.20±15.66	24.87±17.81
	Precision ± (σ)		
	TLR	FL	BL
Configuration 1	59.93±16.94	67.68±13.43	28.12±15.45
Configuration 2	33.69±13.69	70.60±14.08	30.54±14.91
Configuration 3	33.93±20.85	27.97±15.22	72.77±11.19
Configuration 4	29.10±17.84	32.81±17.37	79.94±10.19
Baseline	30.99±16.25	41.90±16.16	20.13±18.61
	F-measure ± (σ)		
	TLR	FL	BL
Configuration 1	54.87±11.61	67.87±9.82	27.55±12.13
Configuration 2	30.62±12.49	70.67±10.78	30.51±12.43
Configuration 3	30.27±18.24	21.50±12.17	76.62±7.13
Configuration 4	23.51±16.50	25.38±12.52	81.59±8.15
Baseline	29.39±12.94	45.98±13.44	15.09±12.36

4.7. ***Research Question 2***

To answer whether there are significant differences in performance among the different configurations of our *FROM* approach and the baseline in Traceability Links Recovery, Feature Location, and Bug Localization, the results should be properly compared. To do this, all of the data resulting from the empirical analysis was analyzed using statistical methods following the guidelines in [43]. The goals of our statistical analysis are: (1) to provide formal and quantitative evidence (statistical significance) that the configurations and the baseline do in fact have an impact on the comparison metrics (i.e., that the differences in the results were not obtained by mere chance); and (2) to show that those differences are significant in practice (effect size).

To enable statistical analysis, all configurations should be run a large enough number of times (independently) to collect information on the probability distribution for each type of query. A statistical test should then be run to assess whether there is enough empirical evidence to claim that there is a difference between the two configurations. In order to do this, two hypotheses are defined: (1) the null hypothesis H_0 is typically defined to state that there is no difference among the configurations and the baseline, and (2) the alternative hypothesis H_1 states that at least one configuration differs from another. A statistical test aims to verify whether H_0 should be rejected.

The statistical tests provide a probability value, *p-value*, which obtains values between 0 and 1. The lower the *p-value* of a test, the more likely that H_0 is false. It is accepted by the research community that a *p-value* under 0.05 is statistically significant [43], and so H_0 can be considered false.

The test to follow depends on the properties of the data. Since our data does not follow a normal distribution, our analysis requires the use of non-parametric techniques. There are several tests for analyzing this kind of data; however, the Quade test shows is more powerful when working with real data [44]. In addition, according to Conover [45], the Quade test has shown better results than the others when the number of algorithms is low (no more than 4 or 5 algorithms).

RQ_2 answer. The *p-Values* and statistics of the Quade test are shown in the upper part of Table 3. Since the *p-Values* shown in this table are smaller than 0.05 in all cases, we reject the null hypothesis. Consequently, we can state that there are significant differences in the configurations and the baseline of Traceability Links Recovery, Feature Location and Bug Localization for all the performance indicators (recall and precision).

4.8. ***Research Question 3***

To answer whether a configuration has a significant impact in performance, the performance of the configuration should be statistically compared against all others. In order to do this, we perform an additional post hoc analysis (pair-wise comparison among configurations, also including the baseline). The middle part of Table 3 shows the *p-Values* of Holm's post hoc analysis for pair-wise comparison of configurations and the baseline for the performance indicators in Traceability Links Recovery, Feature Location and Bug Localization. The majority of the *p-Values* shown in this table are smaller than their

Table 3: Results of the statistical analysis

Quade test statistic and *p-Values*

	Traceability Links Recovery		Feature Location		Bug Localization	
	Recall	Precision	Recall	Precision	Recall	Precision
p-value	$5.0x10^{-16}$	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$
Statistic	21.38	40.42	145.06	110.99	60.81	63.24

Holm's post hoc *p-Values*

	Traceability Links Recovery		Feature Location		Bug Localization	
	Recall	Precision	Recall	Precision	Recall	Precision
C1 vs C2	$2.7x10^{-15}$	$\ll 2x10^{-16}$	0.2	0.15	0.26	0.33
C1 vs C3	$2.7x10^{-08}$	$\ll 2x10^{-16}$	$\ll 2x10^{-16}$	$\ll 2x10^{-16}$	$3.1x10^{-14}$	$3.1x10^{-14}$
C1 vs C4	$\ll 2x10^{-16}$	$\ll 2x10^{-16}$	$\ll 2x10^{-16}$	$\ll 2x10^{-16}$	$4.1x10^{-14}$	$4.1x10^{-14}$
C1 vs Baseline	$4.3x10^{-16}$	$\ll 2x10^{-16}$	$2x10^{-10}$	$\ll 2x10^{-16}$	0.002	0.025
C2 vs C3	0.27	0.8	$\ll 2x10^{-16}$	$\ll 2x10^{-16}$	$9.4x10^{-14}$	$3.1x10^{-14}$
C2 vs C4	0.02	0.03	$\ll 2x10^{-16}$	$\ll 2x10^{-16}$	$4.1x10^{-14}$	$3.1x10^{-14}$
C2 vs Baseline	0.73	0.26	$2.4x10^{-15}$	$\ll 2x10^{-16}$	$9.9x10^{-05}$	0.005
C3 vs C4	0.002	0.15	0.22	0.03	0.28	0.003
C3 vs Baseline	0.36	0.52	$\ll 2x10^{-16}$	$9.6x10^{-12}$	$3.1x10^{-14}$	$5.4x10^{-14}$
C4 vs Baseline	0.002	0.46	$\ll 2x10^{-16}$	$6.6x10^{-05}$	$3.1x10^{-14}$	$3.1x10^{-14}$

$\hat{A}_{12}$ statistic for each pair

	Traceability Links Recovery		Feature Location		Bug Localization	
	Recall	Precision	Recall	Precision	Recall	Precision
C1 vs C2	0.7982	0.8811	0.4456	0.4358	0.4563	0.4388
C1 vs C3	0.7105	0.8312	0.9868	0.9701	0.0006	0.0130
C1 vs C4	0.8433	0.8893	0.9807	0.9381	0.0045	0.0028
C1 vs Baseline	0.8024	0.8873	0.7275	0.8867	0.7018	0.6590
C2 vs C3	0.4487	0.5212	0.9905	0.9736	0.0130	0.0040
C2 vs C4	0.6040	0.5932	0.9839	0.9473	0.0130	0.0005
C2 vs Baseline	0.4812	0.5554	0.7639	0.9066	0.7415	0.6910
C3 vs C4	0.6284	0.5613	0.4555	0.4198	0.4453	0.3226
C3 vs Baseline	0.5378	0.5277	0.0678	0.2653	0.9972	0.9858
C4 vs Baseline	0.3793	0.4627	0.0890	0.3474	0.9949	0.9977

corresponding significance threshold value (0.05), indicating that the differences of performance between the configurations are significant. However, some values are greater than the threshold, indicating that the differences between those configurations are not significant.

However, when comparing configurations with a large enough number of runs, statistically significant differences can be obtained even if they are so small as to be of no practical value [43]. It is important to assess, through *effect size* measures, if a configuration is statistically better than another one, and if so, the magnitude of the improvement.

For a non-parametric effect size measure, we use Vargha and Delaney's $\hat{A}_{12}$ [46, 47]. $\hat{A}_{12}$ measures the probability that running one configuration yields higher values than running another configuration. If the two configurations are equivalent, then $\hat{A}_{12}$ will be 0.5. For example, $\hat{A}_{12} = 0.7$ means that the first of the pair of configurations would obtain better results in 70% of the runs, and $\hat{A}_{12} = 0.3$ means that the second of the pair of configurations would obtain better results in 70% of the runs. We record an $\hat{A}_{12}$ value for every pair of configurations as well as for every configuration and the baseline in Traceability Links Recovery, Feature Location and Bug Localization.

The lower part of Table 3 shows the values of the effect size statistics between the configurations and the baseline in Traceability Links Recovery, Feature Location and Bug Localization. In Traceabil-

ity Links Recovery, the largest differences were obtained in comparisons that entail Configuration 1, where the largest difference is obtained when compared with Configuration 4 (0.8433 for recall and 0.8893 for precision). Therefore, Configuration 1 outperforms Configuration 4 for recall and precision with a pronounced superiority (84.33% of the times for recall and 88.93% of the times for precision). In Feature Location, Configuration 1 and Configuration 2 show a pronounced superiority over Configuration 3, Configuration 4 and the baseline. The largest difference is obtained when comparing Configuration 2 with Configuration 3 (0.9905 for recall and 0.9736 for precision). In Bug Localization, Configuration 3 and Configuration 4 show a pronounced superiority over Configuration 1 and Configuration 2. The largest differences are obtained when comparing Configuration 1 with Configuration 3 for recall (0.0006) and when comparing Configuration 2 with Configuration 4 for precision (0.0005).

RQ_3 answer. From the results, we can conclude that the configuration against the type of query has an actual impact in performance.

5. Discussion

The results of our approach show that the configuration of fitness objectives that provides the best result in Traceability Links Recovery, Bug Localization and Feature Location is different for each of them. As described in Section 4.5, the parameters of the evolutionary algorithm in use have been chosen according to the literature values. However, as suggested by [32] and confirmed in [48], tuned parameters can outperform default values, but are far from optimal in individual problem instances. Since the objective of this paper is to evaluate the different configurations, we do not tune the values to improve the performance of our algorithm.

By analyzing the impact on the results for Traceability Links Recovery, Bug Localization and Feature Location of the four configurations of fitness objectives, our findings suggest that:

1 From the four configurations, there is not a unique combination of objectives that retrieves the best results for all the types of queries.

2 Model Similitude, by itself, obtains the best results in the queries for Traceability Links Recovery but it is not powerful enough to achieve the best results in the queries for Feature Location and Bug Location.

3 Model Understandability, which is a desirable objective (since it allows for an easier comprehension of model fragments by software engineers) cannot be systematically applied to Traceability Links Recovery. The configurations where it is applied (2 and 4) yield worse Traceability Links Recovery results than those where it is not applied.

4 Model Timing is only useful for Bug Localization, not contributing to improve the results in Traceability Links Recovery or Feature Location.

5 Requirements, bugs, and features can all be described through NL, but are of different nature. Different fitness configurations guide the Evolutionary Algorithm better, depending on the task (Traceability Links Recovery, Bug Localization, Feature Location) that is being carried out: Configuration 1 (Similitude) for Traceability Links Recovery, Configuration 4 (Similitude + Understandability + Timing) for Bug Localization, and Configuration 2 (Similitude + Understandability) for Feature Location.

The results of evaluating our approach show that Traceability Link Recovery achieves the worst results. We detected that this happens because when requirements are written, part of the domain knowledge related to the requirements is assumed to be known by all the domain experts, so it is not formalized. For example, given the requirement: *At all stations, the doors are automatically opened*, the engineers understand that the doors have to be opened in all the stations without being requested by a passenger. However, this requirement embodies tacit knowledge that is obvious to the domain engineers: *The train has doors on both sides, but only the doors on the side of the platform will be opened while the doors on the*

side of the tracks will remain closed, and *all the doors of one side will be opened, except the driver's door in the cabin*.

Tacit knowledge is not reflected in the text of the requirements, since it is shared between the engineers who write and read the requirements. As a result, the models are built through both the text of the requirements, and the tacit knowledge of the engineers, leading to models that contain elements built according to the text of the requirements, and elements built through tacit knowledge.

However, since part of the knowledge is not reflected in the text of the requirement, the similitude objective is negatively influenced. The similitude objective establishes the similarity between the query and the model fragment according to the co-occurrences of terms between both. Configuration 1 (similitude objective only) achieves worse results for Traceability Link Recovery than for Feature Location. Feature descriptions are less vulnerable to the tacit knowledge issue since they are written in a different style, in a different moment of the software life cycle, and with a different goal in mind. Requirements play a key role in the contracts between our industrial partner and their clients, but feature descriptions are for internal use only.

Model understandability does not pay off in the particular case of Traceability Link Recovery. Configurations 2 and 4 (which include the understandability objective) achieve worse results than Configuration 1 (similitude objective only). Model understandability favors model fragments that involve a lower number of model elements. In the face of (1) a model fragment (that includes model elements related to the tacit knowledge), and (2) a model fragment that is a subset of the former (without the model elements related to the tacit knowledge), the first model fragment not only does not achieve a better result for the similitude objective, but it is also penalized by the understandability objective because of its higher model elements count.

Also, our results confirm the relevance of the Defect Principle [13] in model fragment retrieval since the configurations that include Model Timing (Configuration 3 and 4) obtained the best results in Bug Location. This is because the majority of bugs (about 90%) provided by our industrial partner are related to recent modifications. In contrast, Model Timing negatively influenced the results in Traceability Links Recovery and Feature Location. Given either a requirement or a feature description, it is not safe to assume than in most of the cases it is related to a model fragment modified recently.

Vocabulary mismatch is a phenomena that occurs when when distinct words are used to refer to the same concept in both query and models. This happens most when the engineer in charge of defining the query (requirement, feature description, or bug description) has not been involved in the construction of the model, and when different engineers are in charge of working with the queries and the models.

Even though we use Natural Language Processing (NLP) to unify the language of the terms shared by queries and models, vocabulary mismatch remains an issue that must be taken into account: the in-house terms are often not recognized as eligible synonyms, and are therefore excluded from NLP, leading to vocabulary mismatch. For example, the terms *PLC* and *system* may be recognized as synonyms, but the terms *PLC* and *COSMOS*[3] are definitely not known to be synonyms, because *COSMOS* is an in-house term that is used exclusively by our industrial partner to refer to the term *PLC*. To minimize the vocabulary mismatch issue, NLP should be extended in order to include a list of in-house synonyms.

Finally, our approach takes as input a query to provide a ranking of solutions that the engineer can inspect instead of having to look for solutions manually. This helps engineers since they do not have to inspect large and complex models manually each time that a software maintenance activity needs to be carried out. The engineers can also consider the solutions of the ranking as a starting point from where solutions can be manually refined. Furthermore, after inspecting the solutions, the user may refine the query and iterate the process to obtain different solutions. Our findings are encouraging and indicate that we should further research this field.

[3] `http://www.cafpower.com/en/systems/control-communication/tcms-system-cosmos`

6. Threats to validity

We follow the guidelines suggested by De Oliveira et al. [49] to identify the threats to the validity of our work.

Conclusion validity threats: The first threat of this type is not accounting for random variation. To address this threat, we considered 30 independent runs for each query and configuration. The second threat is the lack of a formal hypothesis and statistical tests. In this paper we employed standard statistical analysis following accepted guidelines [32] to avoid this threat. The third threat is the lack of a good descriptive analysis. In this work, we have used the recall, precision and F-measure measurements to analyze the confusion matrix obtained; however, other measurements could be applied. Some works argument that the use of the Vargha and Delaney $\hat{A}_{12}$ measurement can be miss-representative [32] and that data should be pre-transformed before applying it. We did not find any use cases for data pre-transformation that applied to our case study.

Internal validity threats: The first identified threat of this type is the poor parameter settings threat. In this paper we used standard values for the algorithms. As suggested by Arcuri and Fraser [32], default values are good enough to measure the performance of location techniques. These values have been tested in similar algorithms for Feature Location [50]. In addition, the tuning of the 'k' value in the application of SVD can affect the results of LSI, and should be further studied [22, 23]. Nevertheless, we plan to evaluate all the parameters of our algorithm in a future work. The second threat is the lack of real problem instances. The evaluation of this paper was applied to an industrial case study.

Construct validity threats: The identified threat is the lack of assessing the validity of cost measures threat. To address this threat we performed a fair comparison among the configurations by allocating a fixed amount of wall clock time for each run of the algorithm in order to set the same amount of time to traverse the search space.

External validity threats: In order to mitigate the lack of a clear object selection strategy, our approach uses an industrial case study, which instances are collected from real world problems.

Moreover, our approach has been designed to be generic and applicable not only to the domain of our industrial partner but also to other different domains: the fitness function can be applied to any model conforming to MOF, and the text elements associated to the models are extracted automatically by the approach using the reflective methods provided by the Eclipse Modeling Framework. The requisites to apply our approach are that the set of models conform to MOF, and the query is provided in NL. However, our approach should be applied to other domains before assuring its generalization.

7. Related Work

Works related to this one comprehend Traceability Links Recovery, Bug Localization, and Feature Location. Through this section, some of these related works are analyzed and compared with ours.

7.1. ***Traceability Links Recovery***

There are several approaches to Traceability Links Recovery, being NLP and LSI the most common. The role of NLP in requirements engineering is vital to the Software Engineering community [51]. NLP has been applied to tackle Traceability Links Recovery at several levels of abstraction and specific problems and tasks in works like [52, 53] or [54]. In [55], NLP is used to identify equivalence between requirements, and a series of performance evaluation principles to do so are defined. The authors conclude that the performance of NLP is determined by the properties of the studied dataset. They measure the properties as a factor to adjust NLP, and apply their principles to an industrial case study. The work presented in [56] uses NLP to study how changes in requirements impact other requirements. The authors analyze Traceability Links Recovery between requirements, and use NLP to determine how changes in requirements must propagate. The work presented in [57] uses LSI and analyst feedback to trace code to requirements. Finally, the authors of [58] consider the possible configurations of LSI when using the technique for Traceability Links Recovery between requirements and test

cases, and state that LSI configurations depend on the datasets. They look forward to automatically determining said configuration.

Our work differs from [51, 52, 53, 54], since we do not use NLP as a means of Traceability Links Recovery analysis. We do not evaluate its performance nor the tweaking of NLP as [55] does. Instead, we use NLP to unify the input for LSI. In addition, our work also differs from [56], since we do not tackle changes in requirements nor Traceability Links Recovery between requirements, but rather study Traceability Links Recovery between requirements and a set of evolving model fragments. Moreover, our work also differs from [57] since we do not use feedback from humans in the tracing process and we target models instead of code. In contrast with [58], we do not tackle LSI configurations or their impact on Traceability Links Recovery, but rather analyze how different fitness objectives configurations affect the Evolutionary Algorithm when recovering traceability links.

7.2. ***Bug Localization***

In recent years, many Bug Localization approaches have been proposed. Lukins et al. [59] used Latent Dirichlet Allocation (LDA) for predicting the location of a newly reported bug through source code comments and identifiers as information resources. Zhou et al. [60] proposed a revised Vector Space Model (VSM) approach for improving the performance of bug localization, based on the idea that bugs are more likely to appear in larger files, also using the similarity between the text of new bug reports and previously fixed bugs. Thomas et al. [22] evaluated the performance of combinations of IR-based classifiers for bug location in code. Saha et al. [61] presented BLUiR, which uses a TF-IDF model baseline. They believe code constructs improve the accuracy of bug localization, so the source code is syntactically parsed into four document fields: class, method, variable, and comment. The summary and the description of a bug report are considered as query fields. Textual similarities are computed for each of the eight document-query pairs, and summarized into a ranking. Kim et al. [62] propose a one-phase and a two-phase prediction models to recommend files to fix. In the one-phase model, they create features from textual information and meta-data, apply Naïve Bayes to train the model using fixed files as classification labels, and use said model to assign source files to a bug report. In the two-phase model, they apply their one-phase model to classify a new bug report as "predictable" or "deficient", and then make predictions for "predictable" reports. These approaches target code, while our approach targets models to locate the bug realizations. Moreover, these approaches rely on IR techniques only, while ours uses an Evolutionary Algorithm that generates possible solutions. In addition, we tackle how different combinations of objectives affect the results of our approach.

Zamani et al [63] proposed an approach to rank source code locations, based on textual similarity with change requests and the use of time meta-data. This approach gives better results than IR techniques, however, it is applied at the source code level. We use a Evolutionary Algorithm to address the location of bugs in models. In our case, the Defect principle is one of the three computed objectives, activated depending on the configuration.

7.3. ***Feature Location***

Approaches related to Feature Location comprehend feature and requirement location techniques. Typechef [64] provides an infrastructure to locate code associated to a given feature by analyzing #ifdef directives. Trace analysis [65] is a technique that indicates the executed code at run-time. Some approaches related to feature location use LSI to extract code associated to a feature [7, 66]. These techniques have been generally applied to search code. In contrast, our approach searches for model fragments.

Feature location approaches in product families [67] center their efforts in finding the code that implements a feature between different products through FCA [68] and LSI. In our approach, we are instead interested in locating the most relevant model fragments for a feature. Other works [69] focus on applying reverse engineering to source code to obtain the variability model. In [70] the authors use propositional logic to describe the dependencies between

features. In [71] the authors combine Typechef and propositional logic to extract conditions among features. These works engage the variability of products, but do tackle the most relevant model fragments for the development of features.

In [72], Lapeña et al. use POS Tagging along with an adapted two-step LSI to obtain rankings of methods for the requirements of a new product in a product family. In the presented work, we use a Multiple Objective Evolutionary Algorithm (MOEA) to find model fragments that can be used to implement a particular feature, and analyze how distinct fitness objective configurations affect the results instead.

Some works [73, 74, 75, 76, 77] focus on the location of features over models by comparing the models with each other to formalize the variability among them. The presented work differs from these works in that the aim is not to formalize the variability, but to locate model fragments relevant to the provided feature descriptions.

Font et al. [17] use a Single Objective Evolutionary Algorithm (SOEA) to locate features among a family of models. Their approach is refined in [18], where the authors use a SOEA to find sets of suitable feature realizations. The authors cluster model fragments based on their common attributes through FCA, and then LSI ranks the candidates based on the similarity with the feature description. The presented approach differs from [17] and [18] by leveraging a MOEA, with a fitness function that combines three fitness objectives to determine the fitness scores of the evolving model fragments.

In [78], Font et al. performed a comparison between five different SOEAs (Evolutionary Algorithm, Random Search, Steepest Ascent Hill Climbing with Replacement, Iterated Local Search with Random Restarts, and Hybrid between Evolutionary and Hill-Climbing) for feature location in models, showing that the best results where achieved by a hybrid between an evolutionary algorithm and a hill climbing. Cetina et al. [79] explored a new direction: taking advantage of already long-living software systems (designed with sustainability in mind) to address the challenge of feature location. Specifically, they used commonality and modifications fitness though model retrospectives in order to promote model fragments that suffered less modifications throughout time. Through this work, we analyze the impact that different configurations of objectives (four combinations of similitude, understandability and timing) have over the results depending on the maintenance task that is performed, which is something that [17, 18, 78, 79] do not tackle.

8. Concluding Remarks

Traceability Links Recovery, Bug Localization, and Feature Location are amongst the most common tasks in the Software Engineering field. However, their application to conceptual models has not received enough attention yet. We propose an approach, named Fragment Retrieval on Models (*FROM*), that uses a Multi-Objective Evolutionary Algorithm to retrieve the most relevant model fragments for different types of queries (NL requirements for Traceability Links Recovery, bug descriptions for Bug Localization, and feature descriptions for Feature Location). Our approach is guided by four configurations that combine three different fitness objectives: Model Similitude, Model Understandability, and Model Timing. Through this work, we analyze the impact of each configuration on the results of the Evolutionary Algorithm for Traceability Links Recovery, Bug Localization, and Feature Location.

Our results show new findings that are relevant for general fragment retrieval approaches since none of the four configurations achieve the best results for all the types of NL queries provided as input. Requirements, bugs and features can be described using NL but depending on the task that is being carried out (Traceability Links Recovery, Bug Localization, Feature Location) different fitness configurations are better. For example, model fragment similitude obtains the best results for Traceability Links Recovery but it is not powerful enough to obtain the best results in Bug Localization, in which model timing is useful to improve the results.

In future iterations of our work, we will perform parameter tuning of the evolutionary algorithm and the dimensions 'k' value of LSI. We also plan to evaluate machine learning techniques, such as the ones of the learning to Rank family [80], as fitness objectives

that guide the retrieval of model fragments for model maintenance tasks.

Acknowledgements

This work has been partially supported by the Ministry of Economy and Competitiveness (MINECO) through the Spanish National R+D+i Plan and ERDF funds under the project Model-Driven Variability Extraction for Software Product Line Adoption (TIN2015-64397-R).

References

[1] R. Oliveto, M. Gethers, D. Poshyvanyk, A. De Lucia, On the Equivalence of Information Retrieval Methods for Automated Traceability Link Recovery, in: IEEE 18th International Conference on Program Comprehension, 2010.

[2] A. Mahmoud, N. Niu, S. Xu, A Semantic Relatedness Approach for Traceability Link Recovery, in: IEEE 20th International Conference on Program Comprehension, 2012.

[3] B. Dit, M. Revelle, M. Gethers, D. Poshyvanyk, Feature Location in Source Code: A Taxonomy and Survey, in: Journal of Software Maintenance and Evolution: Research and Practice, 2011.

[4] J. Rubin, M. Chechik, A Survey of Feature Location Techniques, in: Domain Engineering, Springer, 2013.

[5] T. K. Landauer, P. W. Foltz, D. Laham, An Introduction to Latent Semantic Analysis, Discourse processes 25 (1998).

[6] T. Hofmann, Probabilistic Latent Semantic Indexing, in: Proceedings of the 22nd Annual International ACM/SIGIR Conference on Research and Development in Information Retrieval, 1999.

[7] D. Poshyvanyk, Y. Guéhéneuc, A. Marcus, G. Antoniol, V. Rajlich, Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval, IEEE Trans. Software Eng. 33 (2007).

[8] M. Revelle, B. Dit, D. Poshyvanyk, Using data fusion and web mining to support feature location in software, in: IEEE 18th International Conference on Program Comprehension, 2010.

[9] W. E. Wong, R. Gao, Y. Li, R. Abreu, F. Wotawa, A Survey on Software Fault Localization, Transactions on Software Engineering 42 (2016).

[10] M. Brambilla, J. Cabot, M. Wimmer, Model-Driven Software Engineering in Practice, 1st ed., Morgan & Claypool Publishers, 2012.

[11] J. Chimiak-Opoka, Measuring UML Models Using Metrics Defined in OCL within the SQUAM Framework, MODELS (2011).

[12] H. Störrle, On the Impact of Layout Quality to Understanding UML Diagrams: Size Matters, MODELS (2014).

[13] T. Zimmermann, P. Weisgerber, S. Diehl, A. Zeller, Mining Version Histories to Guide Software Changes, in: Proceedings of the 26th International Conference on Software Engineering, 2004.

[14] B. Sisman, A. C. Kak, Incorporating Version Histories in Information Retrieval Based Bug Localization, in: 9th IEEE Working Conference on Mining Software Repositories, 2012.

[15] C. M. Fonseca, P. J. Fleming, et al., Genetic Algorithms for Multiobjective Optimization: Formulation, Discussion, and Generalization, in: ICGA, volume 93, Citeseer, 1993, pp. 416–423.

[16] J. Font, L. Arcega, Ø. Haugen, C. Cetina, Building software product lines from conceptualized model patterns, in: Proceedings of the 19th International Conference on Software Product Line, SPLC, 2015, pp. 46–55.

[17] J. Font, L. Arcega, Ø. Haugen, C. Cetina, Feature Location in Model-Based Software Product Lines Through a Genetic Algorithm, in:

Proceedings of the 15th International Conference on Software Reuse: Bridging with Social-Awareness, 2016.

[18] J. Font, L. Arcega, Ø. Haugen, C. Cetina, Feature Location in Models Through a Genetic Algorithm Driven by Information Retrieval Techniques, in: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, 2016.

[19] S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. De Lucia, T. Menzies, Automatic Query Reformulations for Text Retrieval in Software Engineering, in: 35th International Conference on Software Engineering (ICSE), 2013, pp. 842–851.

[20] A. Hulth, Improved Automatic Keyword Extraction Given More Linguistic Knowledge, in: Proceedings of the Conference on Empirical Methods in Natural Language Processing, 2003.

[21] R. B. Bradford, An empirical study of required dimensionality for large-scale latent semantic indexing applications, in: Proceedings of the 17th ACM conference on Information and knowledge management, ACM, 2008, pp. 153–162.

[22] S. W. Thomas, M. Nagappan, D. Blostein, A. E. Hassan, The impact of classifier configuration and classifier combination on bug localization, IEEE Transactions on Software Engineering 39 (2013) 1427–1443.

[23] M. Borg, P. Runeson, A. Ardö, Recovering from a decade: a systematic mapping of information retrieval approaches to software traceability, Empirical Software Engineering 19 (2014) 1565–1616.

[24] S. Khatiwada, M. Tushev, A. Mahmoud, Just enough semantics: An information theoretic approach for ir-based software bug localization, Information & Software Technology 93 (2018) 45–57.

[25] A. E. Hassan, R. C. Holt, The Top Ten List: Dynamic Fault Prediction, in: 21st IEEE International Conference on Software Maintenance, 2005.

[26] D. Wille, S. Holthusen, S. Schulze, I. Schaefer, Interface variability in family model mining, in: Proceedings of the 17th International Software Product Line Conference: Co-located Workshops, 2013, pp. 44–51.

[27] S. Holthusen, D. Wille, C. Legat, S. Beddig, I. Schaefer, B. Vogel-Heuser, Family model mining for function block diagrams in automation software, in: Proceedings of the 18th International Software Product Line Conference: Volume 2, 2014, pp. 36–43.

[28] X. Zhang, Ø. Haugen, B. Moller-Pedersen, Model comparison to synthesize a model-driven software product line, in: Proceedings of the 2011 15th International Software Product Line Conference (SPLC), 2011, pp. 90–99.

[29] X. Zhang, Ø. Haugen, B. Møller-Pedersen, Augmenting product lines, in: Software Engineering Conference (APSEC), 2012 19th Asia-Pacific, volume 1, 2012, pp. 766–771.

[30] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, Y. L. Traon, Bottom-up adoption of software product lines: a generic and extensible approach, in: Proceedings of the 19th International Conference on Software Product Line (SPLC), 2015, pp. 101–110.

[31] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, Y. l. Traon, Automating the extraction of model-based software product lines from model variants (t), in: 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2015, pp. 396–406.

[32] A. Arcuri, G. Fraser, Parameter Tuning or Default Values? An Empirical Investigation in Search-Based Software Engineering, Empirical Software Engineering 18 (2013).

[33] E. Zitzler, L. Thiele, Multiobjective optimization using evolutionary algorithms - a comparative case study, in: Proceedings of the 5th International Conference on Parallel Problem Solving from Nature, 1998, pp. 292–304.

[34] H. Ishibuchi, Y. Nojima, T. Doi, Comparison between single-objective and multi-objective genetic algorithms: Performance comparison and performance measures, in: IEEE International Conference on Evolutionary Computation, 2006, pp. 1143–1150.

[35] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast and elitist multiobjective genetic algorithm: NSGA-II, IEEE Transactions on Evolutionary Computation 6 (2002) 182–197.

[36] A. S. Sayyad, J. Ingram, T. Menzies, H. Ammar, Scalable Product Line Configuration: A Straw to Break the Camel's Back, in: IEEE/ACM 28th International Conference on Automated Software Engineering, 2013.

[37] Ø. Haugen, B. Møller-Pedersen, J. Oldevik, G. K. Olsen, A. Svendsen, Adding Standardized Variability to Domain Specific Languages, in: 12th International Software Product Line Conference, 2008.

[38] J. Kottmann, OpenNLP, 2017. URL: `https://opennlp.apache.org`.

[39] T. Horsmann, N. Erbs, T. Zesch, Fast or Accurate? - A Comparative Evaluation of PoS Tagging Models, in: Proceedings of the International Conference of the German Society for Computational Linguistics and Language Technology, GSCL, 2015, pp. 22–30.

[40] The English (porter2) Stemming Algorithm, 2002. URL: `http://snowball.tartarus.org/algorithms/english/stemmer.html`.

[41] P. Abeles, Efficient Java Matrix Library, 2017. URL: `http://ejml.org/`.

[42] D. Dyer, The Watchmaker Framework for Evolutionary Computation, 2017. URL: `http://watchmaker.uncommons.org/`.

[43] A. Arcuri, L. Briand, A Hitchhiker's Guide to Statistical Tests for Assessing Randomized Algorithms in Software Engineering, Softw. Test. Verif. Reliab. 24 (2014).

[44] S. García, A. Fernández, J. Luengo, F. Herrera, Advanced Nonparametric Tests for Multiple Comparisons in the Design of Experiments in Computational Intelligence and Data Mining: Experimental Analysis of Power, Inf. Sci. 180 (2010).

[45] W. Conover, Practical Nonparametric Statistics, 3. ed ed., Wiley, 1999.

[46] A. Vargha, H. D. Delaney, A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong, Journal of Educational and Behavioral Statistics 25 (2000).

[47] R. J. Grissom, J. J. Kim, "Effect sizes for research: A broad practical approach, Mahwah, NJ: Earlbaum, 2005.

[48] A. Kotelyanskii, G. M. Kapfhammer, Parameter Tuning for Search-Based Test-Data Generation Revisited: Support for Previous Results, in: 14th International Conference on Quality Software, 2014.

[49] M. de Oliveira Barros, A. C. D. Neto, Threats to Validity in Search-based Software Engineering Empirical Studies, Technical Report 0006/2011, 2011.

[50] R. E. Lopez-Herrejon, L. Linsbauer, J. A. Galindo, J. A. Parejo, D. Benavides, S. Segura, A. Egyed, An assessment of search-based techniques for reverse engineering feature models, J. Syst. Softw. 103 (2015) 353–369.

[51] K. Ryan, The Role of Natural Language in Requirements Engineering, in: Proceedings of IEEE International Symposium on Requirements Engineering, 1993.

[52] H. Sultanov, J. H. Hayes, Application of Swarm Techniques to Requirements Engineering: Requirements Tracing, in: 18th IEEE International Requirements Engineering Conference, 2010.

[53] S. K. Sundaram, J. H. Hayes, A. Dekhtyar, E. A. Holbrook, Assessing Traceability of Software Engineering Artifacts, Requirements Engineering 15 (2010).

[54] C. Duan, J. Cleland-Huang, Clustering Support for Automated Tracing, in: Proceedings of the twenty-second IEEE/ACM International Conference on Automated Software Engineering, 2007.

[55] D. Falessi, G. Cantone, G. Canfora, Empirical Principles and an Industrial Case Study in Retrieving Equivalent Requirements via Natural Language Processing Techniques, Transactions on Software Engineering 39 (2013).

[56] C. Arora, M. Sabetzadeh, A. Goknil, L. C. Briand, F. Zimmer, Change Impact Analysis for Natural Language Requirements: An NLP Approach, in: IEEE 23rd International Requirements Engineering Conference, 2015.

[57] J. H. Hayes, S. K. Sundaram, A. Dekhtyar, Advancing candidate link generation for requirements tracing: The study of methods, IEEE Transactions on Software Engineering 32 (2006) 4–19.

[58] S. Eder, H. Femmer, B. Hauptmann, M. Junker, Configuring Latent Semantic Indexing for Requirements Tracing, in: Proceedings of the Second International Workshop on Requirements Engineering and Testing, 2015.

[59] S. K. Lukins, N. A. Kraft, L. H. Etzkorn, Bug Localization Using Latent Dirichlet Allocation, Inf. Softw. Technol. 52 (2010).

[60] J. Zhou, H. Zhang, D. Lo, Where Should the Bugs Be Fixed? - More Accurate Information Retrieval-based Bug Localization Based on Bug Reports, in: Proceedings of the 34th International Conference on Software Engineering, 2012.

[61] R. K. Saha, M. Lease, S. Khurshid, D. E. Perry, Improving Bug Localization using Structured Information Retrieval, in: 28th IEEE/ACM International Conference on Automated Software Engineering, 2013.

[62] D. Kim, Y. Tao, S. Kim, A. Zeller, Where Should We Fix This Bug? A Two-Phase Recommendation Model, IEEE Transactions on Software Engineering 39 (2013).

[63] S. Zamani, S. P. Lee, R. Shokripour, J. Anvik, A Noun-based Approach to Feature Location using Time-Aware Term-Weighting, Information and Software Technology 56 (2014).

[64] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, T. Berger, Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation, in: Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2011.

[65] A. D. Eisenberg, K. D. Volder, Dynamic Feature Traces: Finding Features in Unfamiliar Code, in: 21st IEEE International Conference on Software Maintenance, 2005.

[66] D. Liu, A. Marcus, D. Poshyvanyk, V. Rajlich, Feature Location via Information Retrieval Based Filtering of a Single Scenario Execution Trace, in: Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering, 2007.

[67] Y. Xue, Z. Xing, S. Jarzabek, Feature Location in a Collection of Product Variants, in: 19th Working Conference on Reverse Engineering, 2012.

[68] B. Ganter, R. Wille, Formal Concept Analysis: Mathematical Foundations, Springer Science & Business Media, 2012.

[69] S. She, R. Lotufo, T. Berger, A. Wasowski, K. Czarnecki, Reverse Engineering Feature Models, in: Proceedings of the 33rd International Conference on Software Engineering, 2011.

[70] K. Czarnecki, A. Wasowski, Feature Diagrams and Logics: There and Back Again, in: Proceedings of the 11th International Software Product Lines Conference, 2007.

[71] S. Nadi, T. Berger, C. Kästner, K. Czarnecki, Mining Configuration Constraints: Static Analyses and Empirical Results, in: 36th International Conference on Software Engineering, 2014.

[72] R. Lapeña, M. Ballarín, C. Cetina, Towards Clone-and-Own Support: Locating Relevant Methods in Legacy Products, in: Proceedings of the 20th International Conference on Software Product Lines, 2016.

[73] D. Wille, S. Holthusen, S. Schulze, I. Schaefer, Interface Variability in Family Model Mining, in: 17th International Software Product Line Conference, 2013.

[74] S. Holthusen, D. Wille, C. Legat, S. Beddig, I. Schaefer, B. Vogel-Heuser, Family Model Mining for Function Block Diagrams in Automation Software, in: 18th International Software Product Lines Conference, 2014.

[75] X. Zhang, Ø. Haugen, B. Møller-Pedersen, Model Comparison to Synthesize a Model-Driven Software Product Line, in: Proceedings of the 15th International Conference on Software Product Lines, 2011.

[76] X. Zhang, Ø. Haugen, B. Møller-Pedersen, Augmenting Product Lines, in: 19th Asia-Pacific Software Engineering Conference, 2012.

[77] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, Y. L. Traon, Bottom-up Adoption of Software Product Lines: a Generic and Extensible Approach, in: Proceedings of the 19th International Conference on Software Product Lines, 2015.

[78] J. Font, L. Arcega, Ø. Haugen, C. Cetina, Achieving feature location in families of models through the use of search-based software engineering, IEEE Transactions on Evolutionary Computation (2017).

[79] C. Cetina, J. Font, L. Arcega, F. Pérez, Improving feature location in long-living model-based product families designed with sustainability goals, Journal of Systems and Software 134 (2017) 261 – 278.

[80] T.-Y. Liu, et al., Learning to Rank for Information Retrieval, Foundations and Trends in Information Retrieval 3 (2009) 225–331.

[81] K. Koffka, Principles of Gestalt Psychology, volume 44, Routledge, 2013.

Appendix A. Algorithms

Algorithm 1 Random Fragment Generation

```
1: E ← randomElement(model)
2: F ← newFragment(E)
3: N ← neighbor(E)
4: random ← randomInteger < modelSize
5: iterator ← 0
6: while iterator < random do
7:     P ← E
8:     E ← N
9:     F ← add(E)
10:    N ← neighbor(E) ≠ P
11:    if N = φ then
12:        exit while
13:    end if
14: end while
15: return fragment
```

Algorithm 2 Crossover Operation

```
1: M ← firstParent
2: N ← secondParent
3: F ← fragment(firstParent)
4: if F ∈ N then
5:     I ← individual(F, N)
6:     return I
7: else
8:     I ← individual(F, M)
9:     return I
10: end if
```

Algorithm 3 Mutation Operation

```
1: O ← operation
2: I ← individual
3: if operation = addition then
4:     E ← additionCandidateElement(I)
5:     N ← neighbor(E)
6:     I ← add(N)
7: else
8:     E ← removalCandidateElement(I)
9:     I ← remove(E)
10: end if
11: return I
```

Appendix B. Metrics to measure the size of a model fragment

U1 Counting the number of elements in a model: To do this, [12] uses labels, shapes, and lines. Shapes refer to the visual elements of the models, lines refer to connectors, and labels refer to descriptive independent text. This metric neglects diagram differences, implying that all elements contribute the same amount of complexity and information to the diagram.

U2 Weight factors per model element: In [12], the author uses the findings of [81] to classify the elements in the models into three complexity levels, assign weights accordingly, and computing the diagram size as the weighed number of elements. This metric does not take in account the inherent differences between diagrams.

U3 Weight factors per model element per diagram type: The author of [12] computes the information content of diagram elements (e) as the binary logarithm of the set of elements (E) a modeler may choose from: $weight(e) = \log_2(|E|)$.