Achieving Feature Location in Families of Models through the use of Search-Based Software Engineering

Jaime Font, Lorena Arcega, Øystein Haugen and Carlos Cetina

***Abstract*—The application of Search-Based Software Engineering (SBSE) techniques to new problems is increasing. Feature location is one of the most important and common activities performed by developers during software maintenance and evolution. Features must be located across families of products and the software artifacts that realize each feature must be identified. However, when dealing with industrial software artifacts, the search space can be huge. We propose and compare five search algorithms to locate features over families of product models guided by Latent Semantic Analysis (LSA), a technique that measures similarities between textual queries. The algorithms are applied to two case studies from our industrial partners (leading manufacturers of home appliances and rolling stock) and are compared in terms of precision and recall. Statistical analysis of the results is performed to provide evidence of the significance of the results. The combination of an evolutionary algorithm with LSA can be used to locate features in families of models from industrial scenarios such as the ones from our industrial partners.**

***Index Terms*—Feature Location, Families of Models, Evolutionary Algorithm, Search-Based Software Engineering**

I. Introduction

SEARCH-based techniques have been applied successfully to a growing number of engineering problems. Software engineering is concerned with finding near optimal solutions or those that fall within a specified level of acceptable tolerance. It is precisely these factors which make search-based techniques readily applicable. Search-Based Software Engineering (SBSE) has had notable successes and there is an increasingly widespread application of SBSE across the full spectrum of Software Engineering activities and problems [1]. In addition, some research efforts demonstrate that SBSE, which was previously only applied to laboratory programs, can scale to real-world systems of tens of thousands of lines of code [2].

Feature location (FL) is one of the most important and common activities performed by developers during software maintenance and evolution [3]. FL is known as the process of finding the set of software artifacts that realize a specific feature, and it has received much attention during recent years [4], [5]. However, most of the research on FL targets code as the software artifacts that realize the feature [4], [5], neglecting other software artifacts such as the models. When performing FL over models, the set of possible realizations for a specific feature is too big to be evaluated exhaustively (a model of 500 elements can yield around 10^{29} potential fragments) and there is a need for search-based techniques to drive the process [6].

In this paper, we propose and compare five search algorithms to locate features over a family of models (MFL): Evolutionary Algorithm (EA-MFL), Random Search (RS-MFL), steepest Hill Climbing (HC-MFL), Iterated Local Search with restarts (IHL-MFL), and a hybrid between Evolutionary algorithm and Hill Climbing (EHC-MFL). The five algorithms rely on Latent Semantic Analysis (LSA) [7] as the fitness function for the evaluation of the solutions. LSA is an Information Retrieval technique that measures the similarity between two textual queries, and, in this paper, it is applied to compare the feature realization solutions with the search query that describes the feature being located.

We have applied the five approaches to two different industrial domains: BSH, the leading manufacturer of home appliances in Europe; and CAF, a worldwide leading company that manufactures rolling stock. To compare the search algorithms, we extracted two case studies from our industrial partners that include the problem (the features to be located) and the oracle (the realization of those features validated by the company). Then we compared the results from the five algorithms with the oracle (which is considered to be the ground truth) in terms of precision, recall, the F-measure, and the Matthews Correlation Coefficient (MCC) [8], [9]. Finally, we performed a statistical analysis of the results (following the guidelines in [10]) in order to provide quantitative evidence of the impact of the five search algorithms and to show that this impact is significant.

The EHC-MFL algorithm performed better than the other algorithms in terms of the four performance indicators. For the best case study, up to 72.41% of the model elements that were expected to be in the features being located (according to the oracle) were present when the EHC-MFL was used (up to 67.32% for EA-MFL, 61.81% for ILS-MFL, 56.61% for HC-MFL, and 38.91% for RS-MFL). In addition, only 23.53% of the elements introduced by the EHC-MFL algorithm as part of the feature realization were misplaced and should not be part of it (29.32% for EA-MFL, 39.1% for ILS-MFL, 48.01% for HC-MFL, and 61.67% for RS-MFL). It turns out that the genetic operations performed by EHC-MFL and EA-MFL together with the fitness function are able to properly traverse search spaces that are originated when locating features in

J. Font, L. Arcega, and C. Cetina are with the SVIT Research Group, Universidad San Jorge, Spain, e-mail: (jfont,larcega,ccetina)@usj.es.

J. Font, and L. Arcega are with the Department of Informatics, University of Oslo, Norway.

Ø. Haugen is with the Department of Information Technology, Østfold University College, Norway.

Manuscript received April 19, 2005; revised August 26, 2015.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TEVC.2017.2751100, IEEE Transactions on Evolutionary Computation

industrial models such as the ones from our industrial partners.

The rest of the paper is structured as follows: Section II provides some background. Section III provides an overview of the work. Section IV presents the encoding that was used for the model fragments. Section V shows the usage of LSA as fitness function. Section VI presents the five search algorithms. Section VII presents the evaluation that was performed with our industrial partners and the statistical analysis of the results. Section VIII presents some related work, and then we conclude the paper.

II. Background

This section presents the Domain-Specific Language (DSL) used by our industrial partner BSH to formalize their products, the IHDSL. It will be used throughout the rest of the paper to present a running example. Then the Common Variability Language (CVL) is presented. CVL is the language used by our MFL approaches to formalize the location of the features.

A. The Induction Hobs Domain-Specific Language (IHDSL)

The newest Induction Hobs (IHs) feature full cooking surfaces, where dynamic heating areas are automatically generated and activated or deactivated depending on the shape, size, and position of the cookware placed on top. In addition, there has been an increase in the type of feedback provided to the user while cooking, such as the exact temperature of the cookware, the temperature of the food being cooked, or even real-time measurements of the actual consumption of the IH. All of these changes are made possible at the expense of increasing the software complexity.

The Domain-Specific Language used by our industrial partner to specify the Induction Hobs (IHDSL) is composed of 46 meta-classes, 74 references, and more than 180 properties. However, in order to increase legibility and due to intellectual property rights, we show a simplified subset of the IHDSL at the top of Fig. 1.

Inverters are in charge of transforming the input electric supply to match the specific requirements of the IH. Then, the energy is transferred to the inductors through the channels. There can be several alternative channels, which enable different heating strategies depending on the cookware placed on top of the IH at run-time. The path followed by the energy through the channels is controlled by the power manager. Inductors are the elements where the energy is transformed into an electromagnetic field. Inductors can be organized into groups to heat larger cookware while sharing the user interface.

B. The Common Variability Language applied to IHs

Our MFL approaches use the Common Variability Language (CVL) [11] due to its expressiveness to properly formalize the feature realizations in terms of model fragments. CVL defines variants of a base model that conforms to Meta-Object Facility (MOF) [12] by replacing variable parts of the base model with alternative model replacements that are found in a library.

The base model is a model described by a given DSL (here, IHDSL) that serves as the base for different variants

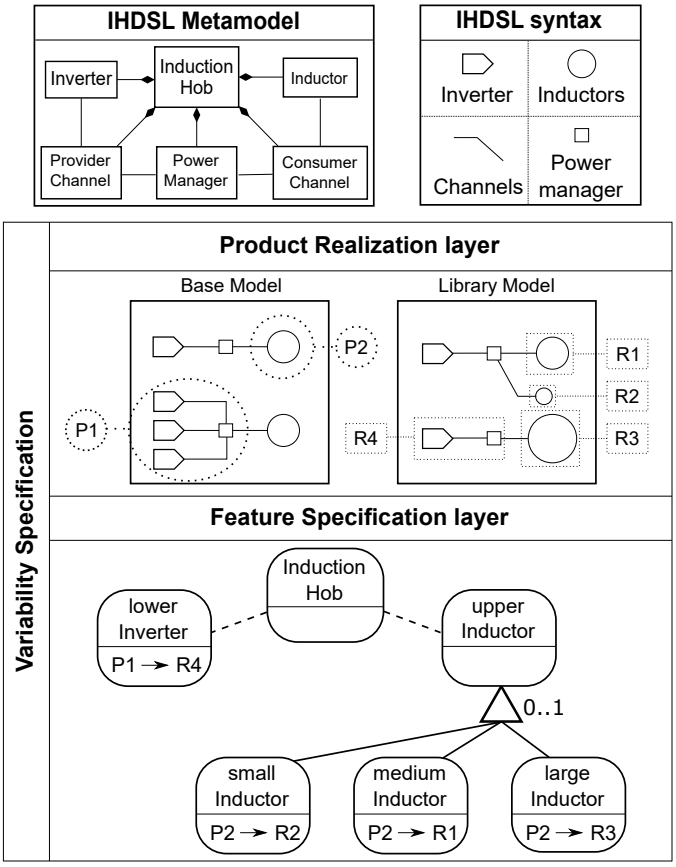


Fig. 1. CVL applied to IHDSL

defined over it. In CVL, the elements of the base model that are subject to variations are the placement fragments (hereafter *placement*). A placement can be any element or set of elements that is subject to variation. To define alternatives for a placement, we use a replacement library, which is a model that is described in the same DSL as the base model that will serve as a base to define alternatives for a placement. Each one of the alternatives for a placement is a replacement fragment (hereafter *replacement*). Similarly to placements, a replacement can be any element or set of elements that can be used as variation for a placement.

The bottom of Fig. 1 shows an example of a variability specification of an IH through CVL. In the product realization layer, two placements are defined over the IH base model (P1 and P2). Then, four replacements are defined over the IH library model (R1, R2, R3, and R4). In the feature specification layer, a Feature Model that formalizes the variability among the IH (based on the placements and replacements) is defined. For instance, P1 can only be substituted by R4 (which is optional), but P2 can be replaced by R1, R2, or R3.

III. Overview

In this paper, we present and compare five search algorithms for Model-based Feature Location (MFL). The objective of the approach is to find the model fragment (from a given set of product models) that realizes a specific feature being described by the user. This is one of the most important and

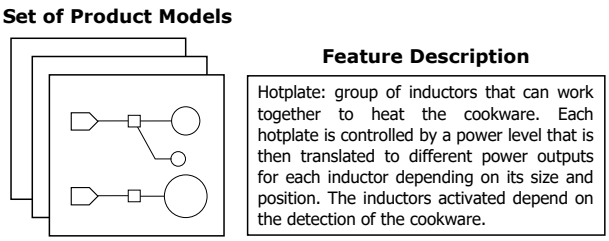


Fig. 2. Input provided to the approach

common activities performed by developers during software maintenance and evolution [3]. This activity takes up to 80% of the time spent on a system [13]. Therefore, increasing the automation level of the feature location activity will help in reducing the time spent on maintenance and evolution tasks.

The five MFL approaches are: Evolutionary Algorithm (EA-MFL); Random Search (RS-MFL); steepest Hill Climbing (HC-MFL); Iterated Local Search with Replacement (ILS-MFL) and a hybrid between Evolutionary Algorithm and Hill-Climbing (EHC-MFL). The same fitness operation is used by the five algorithms. The fitness value is calculated according to the similarity of the model fragment and a feature description that is provided by the user. The outputs of the five approaches are model fragments that may realize the feature being located.

The input for the proposed approaches consists of a set of product models and a textual description of the feature. With these, the engineer can embed their implicit knowledge of the domain into the feature location process.

Fig. 2 presents an example of input for the MFL approaches. The part on the left represents the set of product models. The part on the right shows a textual description for the feature to be located, the hotplate. This is a simplified version of a text description that has been extracted from the internal documentation used by one of our industrial partners to describe their products.

IV. Encoding

The features being located by the algorithms will be in the form of model fragments, which are a subset of the model elements present in a specific product model. Traditionally, evolutionary algorithms encode each possible solution of the problem as a string of binary values. Each position of the string has two possible values: 0 or 1. Therefore, each solution candidate of our proposed approaches will be a model fragment that is defined over one of the product models. Each model element will have a corresponding position in the binary string and the value of that position will indicate the presence or absence of that specific element in the encoded model fragment.

Fig. 3 shows two examples of our encoding of model fragments. We tag each model element of the product model with a letter. In the example of the upper part of Fig. 3, the letters A and F correspond to inverters, the letters B, D, G, and I correspond to channels, and the letters E and J correspond to inductors. The string of binary values that represents the model fragment from this product model has the positions that correspond to each letter with a value of 0 or 1. If the model element is part of the model fragment, the value will

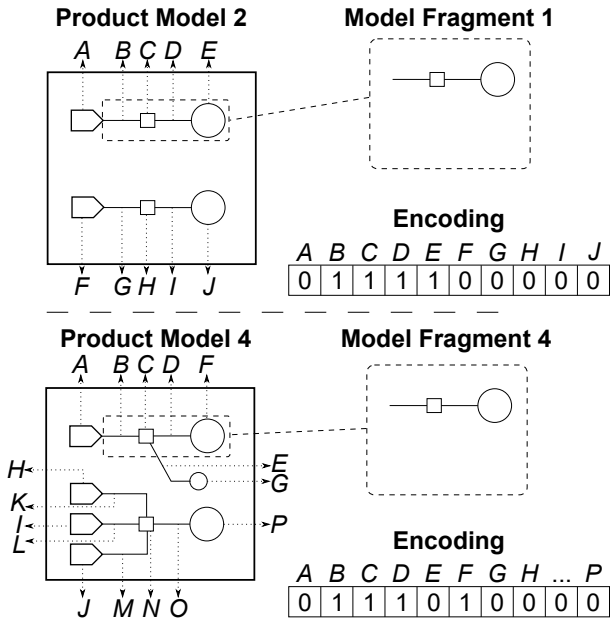


Fig. 3. Encoding of Individuals (Model Fragments)

be 1; if the model element is not part of the model fragment, the value will be 0.

Each model fragment representation depends on the product model that it came from. Both of the examples in Fig. 3 represent the same model fragment, but they come from different product models and thus have different representations. Throughout the rest of the paper, we will refer to each individual as a model fragment that is part of a product model.

V. Model Fragment Fitness

To assess the relevance of each model fragment in relation to the feature description provided by the user, we apply methods that are based on Information Retrieval (IR) techniques. Specifically, we apply Latent Semantic Analysis (LSA) [7] to analyze the relationships between the description of the feature provided by the user and the model fragments previously obtained. Recent studies have shown that there is not a statistically significant difference among different IR techniques [14], [15] when applied to software artifacts [16]. Hence, we chose LSA because it provides results that are similar to other IR techniques for software documents.

LSA constructs vector representations of a query and a corpus of text documents by encoding them as a *term-by-document co-occurrence matrix* (i.e., a matrix where each row corresponds to terms, each column corresponds to documents, and the last column corresponds to the query). We use the *term-frequency (tf)* as the term weighting schema to construct the matrix. In other words, each cell holds the number of occurrences of a term (row) inside either a document or the query (column).

In our work, all documents are model fragments (i.e., a document of text is generated from each of the model fragments). The text of the document corresponds to the names and values of the properties and the methods of each model fragment. The query is constructed from the terms that appear

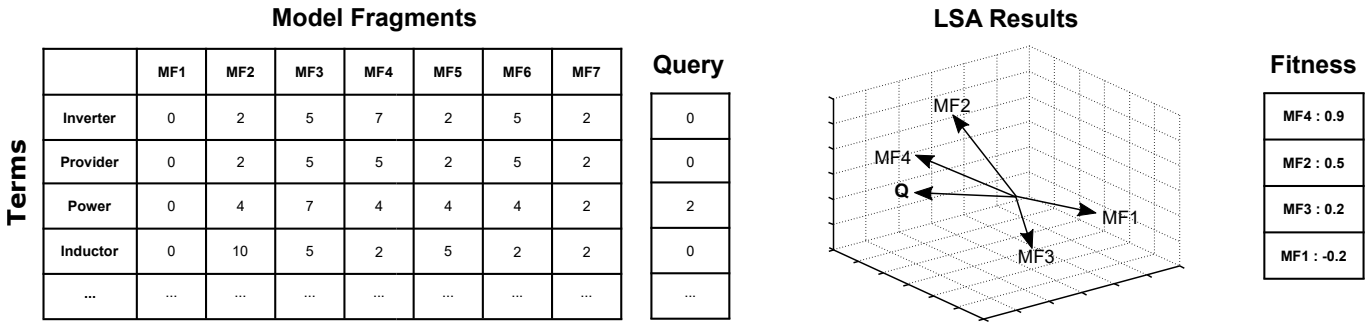


Fig. 4. Fitness Operation via Latent Semantic Analysis (LSA) - term-by-document co-occurrence matrix (left) and vector representation (right)

in the feature description. The text from the documents (model fragments) and the text from the query (feature description) are homogenized by applying well-known Natural Language Processing techniques:

First, the textual description is **tokenized** (divided into words). Usually, a white space tokenizer can be applied (which splits the strings whenever it finds a white space). However, for some sources of description, more complex tokenizers need to be applied. For instance, when the description comes from documents that are close to the implementation of the product, some words could be using CamelCase naming.

Second, we apply the **Parts-of-Speech** (POS) tagging technique. POS tagging analyzes the words grammatically and infers the role of each word into the text provided. Recent studies in software engineering have proven the usefulness of POS-tagging techniques to remove textual noise in software documents [17]. In addition, the use of word-selection strategies [18], [19] can improve the results in feature location [20]. After applying this technique, each word is tagged, which allows the removal of some categories that do not provide relevant information. For instance, conjunctions (e.g., *or*), articles (e.g., *a*), or prepositions (e.g., *at*) are words that are commonly used and do not contribute relevant information that describes the feature, so they are removed.

Third, **stemming** techniques are applied to unify the language that is used in the text. This technique consists of reducing each word to its roots, which allows different words that refer to similar concepts to be grouped together. For instance, plurals are turned into singulars (*inductors* to *inductor*) or verbs tenses are unified (*using* and *used* are turned into *use*).

The union of all of the keywords extracted from the documents (model fragments) and from the query (feature description) are the terms (rows) used by our LSA fitness operation.

Fig. 4 (left) shows an example of the co-occurrence matrix for our running example. Each column is one of the model fragments. The last column is the query that is obtained from the feature description of the user. Each row is one of the terms extracted from the corpora of text, which is composed by all of the model fragments and the query itself (to improve readability, we show the terms before the stemming process). Each cell shows the number of occurrences of each of the terms in the model fragments.

Once the matrix is built, we normalize and decompose it into a set of vectors using a matrix factorization technique called Singular Value Decomposition (SVD) [7]. SVD projects the original term-by-document co-occurrence matrix in a lower dimensional space k. We use the value of k suggested by Kuhn et al. [21], which provides good results [22]. One vector that represents the latent semantics of the document is obtained for each model fragment and the query. Finally, the similarities between the query and each model fragment are calculated as the cosine between the two vectors. The fitness value that is given to each model fragment is obtained as the cosine similarity between the two vectors, obtaining values between -1 and 1.

Let p_1 be an individual of the population; let A be the vector representing the latent semantic of p_1; let B be the vector representing the latent semantics of the query where the angle formed by the vectors A and B is θ. The fitness function can be defined as:

$$fitness(p_1) = \cos(\theta) = \frac{A \cdot B}{\|A\| \cdot \|B\|} \quad (1)$$

Fig. 4 (right) shows a three-dimensional graph of the LSA results. The graph shows the representation of each one of the vectors, which are labelled with letters that represent the names of the model fragments. Finally, after the cosines are calculated, we obtain a value for each of the model fragments, indicating its similarity with the query.

VI. Search Algorithms for FML

In this section, we present the five different search algorithms that will be evaluated. All of them will be guided using the same heuristic function, the fitness based on LSA presented in the previous section. Therefore, their differences with each other lie in how the solution space is traversed looking for the best solution.

A. Evolutionary Algorithm MFL (EA-MFL)

Our first search algorithm is an Evolutionary Algorithm that iterates a population of model fragments and evolves them using genetic operations. As output, the algorithm provides the model fragment that best realizes the feature according to the fitness function. The generation of model fragments is done by applying genetic operators that we have adapted to work with model fragments.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TEVC.2017.2751100, IEEE Transactions on Evolutionary Computation

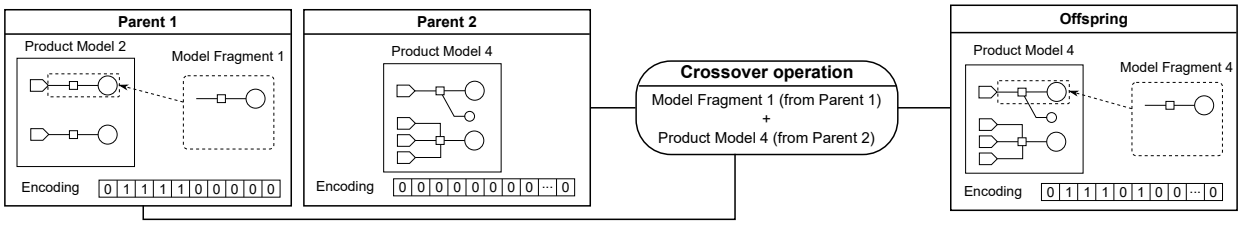


Fig. 5. Example of application of the Crossover Operation on Model Fragments

We develop our evolutionary algorithm as outlined in Algorithm 1 (available in the Appendix). The algorithm starts with a random initial population of model fragments that are obtained from the product models provided as input, (P, Lines 7-14). From this initial population, the offspring of model fragments is created by applying a selection operation (Line 18), a crossover operation (Line 19), and a mutation operation (Line 20). Then, the new offspring is added to the population (Line 21). When the creation of the new population has finished (Lines 17-22), it replaces the existing one (Line 4). This loop (Lines 2-5) is repeated until the $StopCondition$ is met. Below we describe the genetic operations used by this algorithm.

1) Selection of parents: The evolutionary algorithm uses the selection operator to select the best model fragments from the population to be used as the input for the following operations. There are different methods that can be used to perform the selection of the parents, but one of the most common choices is to follow the wheel selection mechanism [23]. In other words, each model fragment from the population has a probability of being selected that is proportional to its fitness score. Therefore, model fragments with high fitness values will have higher probabilities of being chosen as parents for the next generation.

2) Crossover: In our encoding, there are two elements that can be mapped across the different individuals: the model fragment and the referenced product model. Therefore, our crossover operation will take the model fragment from the first parent and the product model from the second parent, generating a new individual that contains elements from both parents, thereby preserving the basic mechanics of the crossover operation.

To achieve the above, our crossover operation is based on model comparisons. Fig. 5 shows an example of the application of the crossover operation on model fragments. First, we select the model fragment from the first parent. Then, we select the product model from the second parent. The model fragment (from first parent) is then compared with the product model (from the second parent). If the comparison finds the model fragment in the product 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. In the case that the comparison does not find a similar element, the crossover will return the first parent unchanged.

This operation enables the search space to be expanded to a different product model, i.e., both model fragments (the one from the first parent and the one from the new individual) will be the same. However, since each of them is referencing a different product model, they will mutate differently and provide different individuals in further generations.

3) Mutation: Fig. 6 shows an example of our mutation for model fragments. Each model fragment is associated to a product model, and the model fragment mutates in the context of its associated product model. In other words, the model fragment will gain or drop some elements, but the resulting model fragment will still be part of the referenced product model. The mutation possibilities of a given model fragment are driven by its associated product model. All the potential model fragments for a given product model can be achieved through the combination of additive and subtractive mutations (given that the model fragment is a subset of the product model elements); therefore there is no need for more complex mutation operations, such as the *change mutation* (a combination of a subtractive and an additive mutation).

To perform the mutation, the type of mutation that will occur (either the addition or removal of elements) is decided randomly:

Subtractive Mutation: This kind of mutation randomly removes an element from the model fragment. Since the resulting model fragment is a subset of the original model fragment and the original is part of the referenced product model, the resulting product model will always be part of the

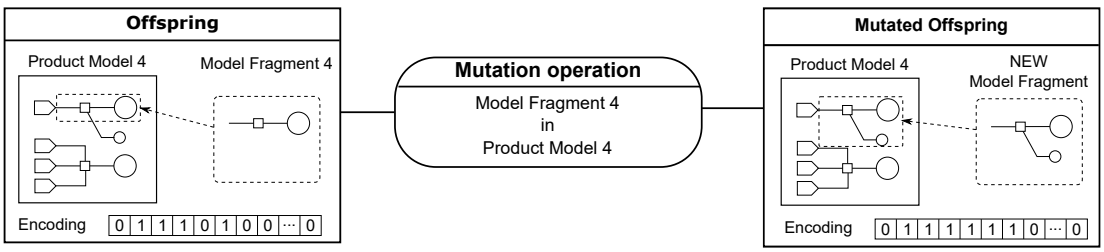


Fig. 6. Example of application of the Mutation Operation on Model Fragments

referenced product model.

Additive Mutation: This kind of mutation randomly adds some elements to the model fragment. The only constraint is that the resulting model fragment be part of the referenced product model. To achieve this, the boundaries of the model fragment with the rest of the product model are identified and a random element that is connected to any of those boundaries (and that is not currently part of the model fragment) is added to the resulting model fragment (See Fig. 6: the inverter and the inductor are connected to the boundaries and could be added as part of the additive mutation). By doing so, the mutated model fragment will be part of the referenced product model.

As a result, a new model fragment is created, but it still references the same product model. In other words, the individual represents other possible feature realizations (that are part of the product model) for the specific feature being located.

B. Random Search MFL (RS-MFL)

The second search algorithm that we use is a standard random search that is used as a sanity check. We want to determine if the presented algorithms perform better than mere chance (represented by this Random Search algorithm).

We used this algorithm as outlined in Algorithm 2 (available in the Appendix). The algorithm starts with a random initial model fragment, ($Best$, Line 1). Then, a new random model fragment is generated (Line 4). The search moves to a new model fragment if the fitness value is better than the current $Best$ model fragment (Lines 5-6). This loop (Lines 3-9) is repeated until the $StopCondition$ is met.

C. Hill Climbing MFL (HC-MFL)

The third search algorithm that we use is Steepest Ascent Hill Climbing with Replacement [24]. The algorithm starts with a random individual and then evaluates its neighbors (small modifications of the individual) looking for a better one (in terms of the fitness function). Those steps are repeated until the stop condition is met. The algorithm does not count with random restarts; therefore, the quality of the solution found by the algorithm greatly depends on the first individual created.

We used this algorithm as outlined in Algorithm 3 (available in the Appendix). The initial model fragment is generated randomly (S, Line 1). A neighborhood of size $NSize$ is created based on this initial model fragment by applying the *tweak* operation (Lines 7-13). This operation applies the mutation operation described in Section VI-A3 and keeps track of the best neighbor of the group (Lines 9-11). After exploring the neighborhood, if a neighbor has a better fitness value than the current state, the search moves to that model fragment (Lines 14-16). This algorithm is repeated until the $StopCondition$ is met.

D. Iterated Local Search with Random Restarts (ILS-MFL)

The fourth search algorithm that is used is the Iterated Local Search with Random Restarts. Different versions of this algorithm have been used since 1981 ([25], [26]) and can be seen as an evolution of the Hill-Climbing algorithm. The algorithm uses hill-climbing strategy to search for local optimum during a certain period of time. Then, it switches to a near hill (restart) and performs hill climbing on the new hill. The main particularity of this search algorithm is the selection of a new hill (which is performed by the $newHomeBase$ function).

We used this algorithm as outlined in Algorithm 4 (available in the Appendix). The initial model fragment is generated randomly ($Current$, Line 1). A random distribution of times is then generated (Line 2), splitting the budget into time slots for each local search. Then, solutions obtained through the $tweak$ function will be explored during the available time (Lines 7-12). When the time is over, we store the new $Best$ solution if it is better (Lines 13-15) and check if we have to switch to another $Home$. The $newHomeBase$ function will determine wether or not we need to move based on the fitness difference between the two individuals (Line 16). Then, the individual will undergo a big change ($perturb$ function), which will be achieved by applying the crossover operation described in Section VI-A2. The algorithm is repeated until the $StopCondition$ is met (Lines 5-18).

E. Hybrid between Evolutionary Algorithm and Hill-Climbing (EHC-MFL)

The fifth algorithm that we will compare is a hybrid between the hill-climbing and the evolutionary algorithms. It will take advantage of the hill-climbing capabilities of searching local optimum values, but it will also make use of the crossover operation to move to different hills (and then perform local searches again through the hill-climbing approach).

We used this algorithm as outlined in Algorithm 5 (available in the Appendix). The initial population of model fragments is generated randomly (P, Line 2). Then, hill-climb is performed for a fixed amount of iterations (Line 6) for each of the individuals in the population (Lines 5-9). If a better solution is found in this process, it is stored in the Best variable (Lines 7-9). Then, when the hill-climb part has finished, we move to a different hill through the $breedPopulation$ operation (Line 11) and repeat the process. The algorithm is repeated until the $StopCondition$ is met (Lines 3-12).

VII. Evaluation

The goal of this paper is to provide answers to the following research questions:

RQ1: Can SBSE techniques driven by LSA be applied to locate features in product models from real industrial scenarios?

RQ2: If so, which evolutionary algorithm produces the best results in terms of solution quality?

This section presents the evaluation that was performed to answer the RQs. It includes a description of the experimental setup, a description of the case studies where we applied the five search algorithms, the results obtained, and the statistical analysis performed.

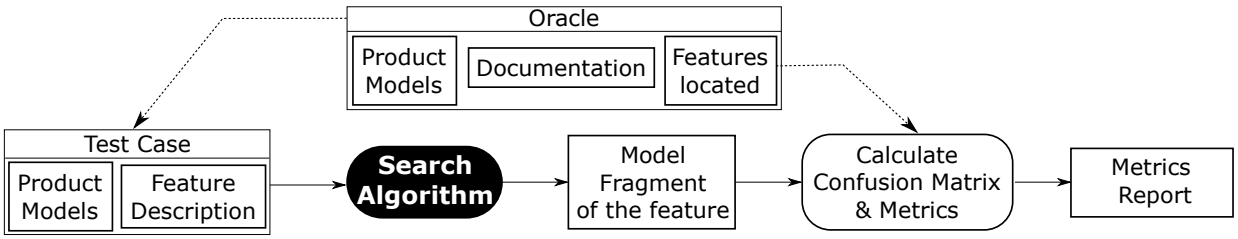


Fig. 7. Experimental Setup

A. Experimental Setup

We use the product models from the case studies as an oracle to evaluate our approach. In other words, we make use of a set of products models whose feature realizations are known beforehand and that will be considered as the ground truth, thus allowing us to compare the results provided by the five algorithms with the oracles.

Fig. 7 shows an overview of the process that was followed to evaluate the five search algorithms (EA-MFL, RS-MFL, HC-MFL, ILS-MFL, and EHC-MFL) for locating features in the two industrial case studies. The top part shows the oracle for the case study: a set of product models, the features located in those product models, and the documentation obtained from our industrial partners. The oracles will be considered the ground truth and will be used to evaluate the five algorithms in terms of MCC, precision, recall, and the F-measure.

First, we elaborated a test case for each feature present in the oracle, including the product models where the feature is present and a textual description obtained from the documentation for the feature. Then, each test case was fed as input for five different algorithms. As a result we obtained a solution in the form of a model fragment for each of the test cases for each algorithm. Those solutions were then compared to the features located from the oracle in order to obtain a confusion matrix.

A confusion matrix is a table that is often used to describe the performance of a classification model (in this case, our algorithms) on a set of test data (the resulting model fragments) for which the true values are known (from the oracle). In our case, each solution outputted by the algorithms is a model fragment that is composed of a subset of the model elements that are part of the product model (where the feature is being located). Since the granularity will be at the level of model elements, the presence or absence of each model element will be considered as a classification. The confusion matrix distinguishes between the predicted values and the real values by 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 metrics are derived from the values in the confusion matrix. Specifically, we will create a report that includes four performance metrics (precision, recall, the F-measure, and the MCC) for each of the test cases for each search algorithm.

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

$$Precision = \frac{TP}{TP + FP} \tag{2}$$

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

$$Recall = \frac{TP}{TP + FN} \tag{3}$$

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

$$F - measure = 2 * \frac{Precision * Recall}{Precision + Recall} = \frac{2 * TP}{2TP + FP + FN} \tag{4}$$

Finally, the MCC is a correlation coefficient between the observed and predicted binary classifications that takes into account all of the observed values (TP, TN, FP, FN) and is defined as follows:

$$MCC = \frac{TP \cdot TN - FP \cdot FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{5}$$

The presented approach has been implemented[1] within the Eclipse environment. We used the Eclipse Modeling Framework [27] to manipulate the models from our industrial partners and the CVL to manage the fragments of models. The evolutionary algorithm was built using the Watchmaker Framework for Evolutionary Computation [28], which allowed us to implement our own genetic operators. The IR techniques that were used to process the language were implemented using OpenNLP [29] for the POS-Tagger and the English (Porter2) [30] as stemming algorithm. Finally, the LSA was implemented using the Efficient Java Matrix Library (EJML [31]).

We performed the execution of the algorithms 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).

[1]The source code can be found here: www.jaimefont.com/flimea.html

TABLE I
AVERAGE TIME REQUIRED TO CONVERGE

	EA-MFL	RS-MFL	HC-MFL	ILS-MFL	EHC-MFL	Budget allocated
Time (s) $\pm$ (σ)	38.3 $\pm$ 10.3	19.9 $\pm$ 11.6	26.5 $\pm$ 9.1	33.7 $\pm$ 8.5	47.4 $\pm$ 12.4	80

1) The BSH case study: The first case study where we applied our approach was BSH (already presented in section II-A as the running example). Their induction division has been producing Induction Hobs under the brands of Bosch and Siemens for the last 15 years.

The oracle extracted from BSH is composed of 46 induction hob models where each product model, on average, is composed of more than 500 elements. The oracle includes 96 different features that can be part of a specific product model. Those features correspond to products that are currently being sold or will be released to the market in the near future. For each of the 96 features, we created a test case that included the set of product models where that feature was used and a feature description; this information was obtained from the documentation of the features in the oracle.

For this case study, we executed 30 independent runs (as suggested by [10]) for each of the 96 test cases for each of the five algorithms (i.e., 96 (features) x 5 (algorithms) x 30 repetitions = 14400 independent runs).

2) The CAF case study: The second case study where we applied our approach was CAF, a worldwide provider of railway solutions. Their trains can be seen all over the world and in different forms (regular trains, subway, light rail, monorail, etc.). A train unit is furnished with multiple pieces of equipment through its vehicles and cabins. These pieces of equipment are often designed and manufactured by different providers, and their aim is to carry out specific tasks for the train. Some examples of these devices are: the traction equipment, the compressors that feed the brakes, the pantograph that harvests power from the overhead wires, or the circuit breaker that isolates or connects the electrical circuits of the train. The control software of the train unit is in charge of making all the equipment cooperate to achieve the train functionality while guaranteeing compliance with the specific regulations of each country.

The DSL of our industrial partner has the required expressiveness to describe the interaction between the main pieces of equipment installed in a train unit. Moreover, this DSL also has the required expressiveness to specify non-functional aspects related to regulation, such as the quality of signals from the equipment or the different levels of installed redundancy.

For instance, the high voltage connection sequence can be described using the DSL. This connection sequence is initiated when the train driver requests its start by using interface devices fitted inside the cabin. The control software is in charge of raising the pantograph to harvest power from the overhead wire and of closing the circuit breaker so the energy can get to converters that adapt the voltage to charge batteries, which, in turn, power the traction equipment.

Again, we extracted an oracle that is composed of 23 trains where, on average, each product model is composed of around 1200 elements. The product models are built using 121 different features that can be part of a specific product model. For each of the 121 features, we created a test case that included the set of product models where that feature was used and a feature description.

For this case study, we executed 30 independent runs for each of the 121 test cases for each of the five algorithms (i.e., 121 (features) x 5 (algorithms) x 30 repetitions = 18150 independent runs). The sum for the two case studies presented is a total of 32550 independent runs.

3) Parameters and Budget: In general, there are two atomic performance measures for search algorithms: one regarding solution quality and one regarding algorithm speed or search effort. In this paper, we focus on the solution quality, trying to determine which algorithm provides solutions that are more similar to the one extracted from the oracle in terms of precision and recall.

Since we allocated a fixed amount of wall clock time for each of the runs of the algorithms, each algorithm has the same amount of time to traverse the search space and so the comparison is fair. First, we run some prior tests to determine the time needed to converge for each of the algorithms, and then we selected the budget time based on those tests.

Then, we use the allocated time to determine the $StopCondition$ of the algorithms (see Table VI and Algorithms in the Appendix). The allocated budget time was 80 seconds (adding a margin to ensure convergence). Although the focus of this paper is the solution quality and not the performance of each algorithm in terms of time, we include the times required by each algorithm to converge as an indication to practitioners when choosing which algorithm to use (see Table I).

As suggested by Arcuri and Fraser [32] and confirmed in Kotelysanskii and Kapfhammer [33], tuned parameters can outperform default values generally, but they are far from optimal in individual problem instances. The focus of this paper is not to tune the values to improve the performance of the algorithms when applied to a specific problem, but rather to compare the performance of the algorithms in terms of solution quality (precision and recall).

Therefore, we will use default parameter values that are commonly used in the literature [34] for the algorithms as described in the literature [32] (see Table VI available on the Appendix). The crossover operation is applied with a probability (p_c) of 0.75. The mutation operation is applied with a probability (mutation probability) of $1/n$ where n is the size of the individual (the number of bits needed to encode that product model). The population size $size$ for the algorithms based on a population will be 100 individuals. Given our crossover operation, the crossover will act over 2 parents (μ) and produce 1 offspring (λ).

TABLE II
MEAN VALUES AND STANDARD DEVIATIONS FOR PRECISION, RECALL, F-MEASURE AND MCC FOR EACH SEARCH ALGORITHM AND EACH CASE STUDY

	BSH				CAF			
	Precision $\pm$ (σ)	Recall $\pm$ (σ)	F-measure $\pm$ (σ)	MCC $\pm$ (σ)	Precision $\pm$ (σ)	Recall $\pm$ (σ)	F-measure $\pm$ (σ)	MCC $\pm$ (σ)
EA	70.68 $\pm$ 14.91	67.32 $\pm$ 14.32	67.34 $\pm$ 11.01	0.58 $\pm$ 0.16	68.80 $\pm$ 14.97	65.81 $\pm$ 14.35	65.91 $\pm$ 11.29	0.59 $\pm$ 0.16
RS	38.33 $\pm$ 16.61	28.12 $\pm$ 14.82	29.21 $\pm$ 13.64	0.20 $\pm$ 0.20	34.18 $\pm$ 14.39	38.91 $\pm$ 15.16	33.74 $\pm$ 12.09	0.28 $\pm$ 0.30
HC	47.07 $\pm$ 15.46	39.48 $\pm$ 13.06	40.26 $\pm$ 10.50	0.28 $\pm$ 0.17	51.99 $\pm$ 14.18	56.61 $\pm$ 16.38	51.97 $\pm$ 11.07	0.45 $\pm$ 0.15
ILS	60.90 $\pm$ 14.76	59.92 $\pm$ 14.07	58.91 $\pm$ 11.16	0.47 $\pm$ 0.19	58.86 $\pm$ 16.89	61.81 $\pm$ 16.57	58.23 $\pm$ 13.75	0.48 $\pm$ 0.22
EHC	76.47 $\pm$ 13.39	72.41 $\pm$ 13.79	72.99 $\pm$ 9.35	0.67 $\pm$ 0.13	71.75 $\pm$ 12.54	67.96 $\pm$ 15.07	68.34 $\pm$ 10.24	0.62 $\pm$ 0.13

B. Statistical Analysis

To properly compare the five algorithms, all of the data resulting from the empirical analysis was analyzed using statistical methods following the guidelines in [10].

In order to answer RQ2, we performed statistical analysis to: (1) provide formal and quantitative evidence (statistical significance) that the five search-based techniques 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) show that those differences are significant in practice (effect size).

1) Statistical significance: To enable statistical analysis, all of the algorithms should be run a large enough number of times (in an independent way) to collect information on the probability distribution for each algorithm. A statistical test should then be run to assess whether there is enough empirical evidence to claim (with a high level of confidence) that there is a difference between the two algorithms (e.g., A is better than B). In order to do this, two hypotheses, the null hypothesis H_0 and the alternative hypothesis H_1, are defined. The null hypothesis H_0 is typically defined to state that there is no difference among the algorithms, whereas the alternative hypothesis H_1 states that at least one algorithm differs from another. In such a case, a statistical test aims to verify whether the null hypothesis H_0 should be rejected.

The statistical tests provide a probability value, $p - value$. The $p - value$ receives values ranging between 0 and 1. The lower the $p - value$ of a test, the more likely that the null hypothesis is false. It is accepted by the research community that a $p - value$ under 0.05 is statistically significant [10], so the hypothesis H_0 can be considered false.

The test that we must follow depends on the properties of the data. Since our data does not follow a normal distribution in general, our analysis requires the use of non-parametric techniques. There are several tests for analyzing this kind of data; however, the Quade test is more powerful than the rest when working with real data [35]. In addition, according to Conover [36], the Quade test has shown better results than the others when the number of algorithms is low, (no more than 4 or 5 algorithms).

However, it is not possible to answer the following question with the Quade test: *Which of the algorithms gives the best performance?*. In this case, the performance of each algorithm should be individually compared against all of the other alternatives. In order to do this, we perform an additional post hoc analysis. This kind of analysis performs a pair-wise comparison among the results of each algorithm, determining whether statistically significant differences exist among the results of a specific pair of algorithms. Specifically, we apply the Holm Post Hoc procedure, as suggested by Garcia et. al. [35].

2) Effect size: When comparing algorithms 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 [10]. Then it is important to assess if an algorithm is statistically better than another and to assess the magnitude of the improvement. *Effect size* measures are used to analyze this.

For a non-parametric effect size measure, we use Vargha and Delaney's $\hat{A}_{12}$ [37], [38]. $\hat{A}_{12}$ measures the probability that running one algorithm yields higher performance values than running another algorithm. If the two algorithms are equivalent, then $\hat{A}_{12}$ will be 0.5.

For example, $\hat{A}_{12} = 0.7$ means that we would obtain better results 70% of the times with the first of the two algorithms compared, and $\hat{A}_{12} = 0.4$ means that we would obtain better results 60% of the times with the second of the two algorithms. Thus, we have an $\hat{A}_{12}$ value for every pair of algorithms.

C. Results

Fig. 8 presents the mean values of precision, recall, the F-measure and the MCC for each feature located for the two case studies and the five algorithms. The first column of the charts (see Fig. 8a ,Fig. 8c, Fig. 8e, 8g, 8i) shows the results for the BSH case study, and the second column of the charts shows the results for the CAF case study (see Fig. 8b, 8d, 8f, 8h, 8j). The first row shows the results for EA-MFL, the second row shows the results for RS-FML, the third row shows the results for HC-MFL, the fourth row shows the results for ILS-MFL, and the fifth row shows the results for EHC-MFL. Each point in the charts represents the mean value (between the 30 independent runs) of the two performance indicators (precision on the x axis and recall on the y axis) for one of the features located for that case study and algorithm.

In Table II, we outline the results aggregated for each algorithm and case study. We also show the F-measure and the MCC performance indicators. The EHC-MFL algorithm achieves the best results for all the performance indicators, providing a precision value of 76.47% in the BSH case study and a precision value of 71.75% in the CAF case study. The Recall achieved is 72.41% for BSH and 67.96% for CAF. The combined F-measure is 72.99% for BSH and 68.34% for CAF. Finally, the MCC achieved is 0.67 for BSH and 0.62 for CAF.

The EA-MFL algorithm follows EHC-MFL and is the second best option, providing values around 10% lower than

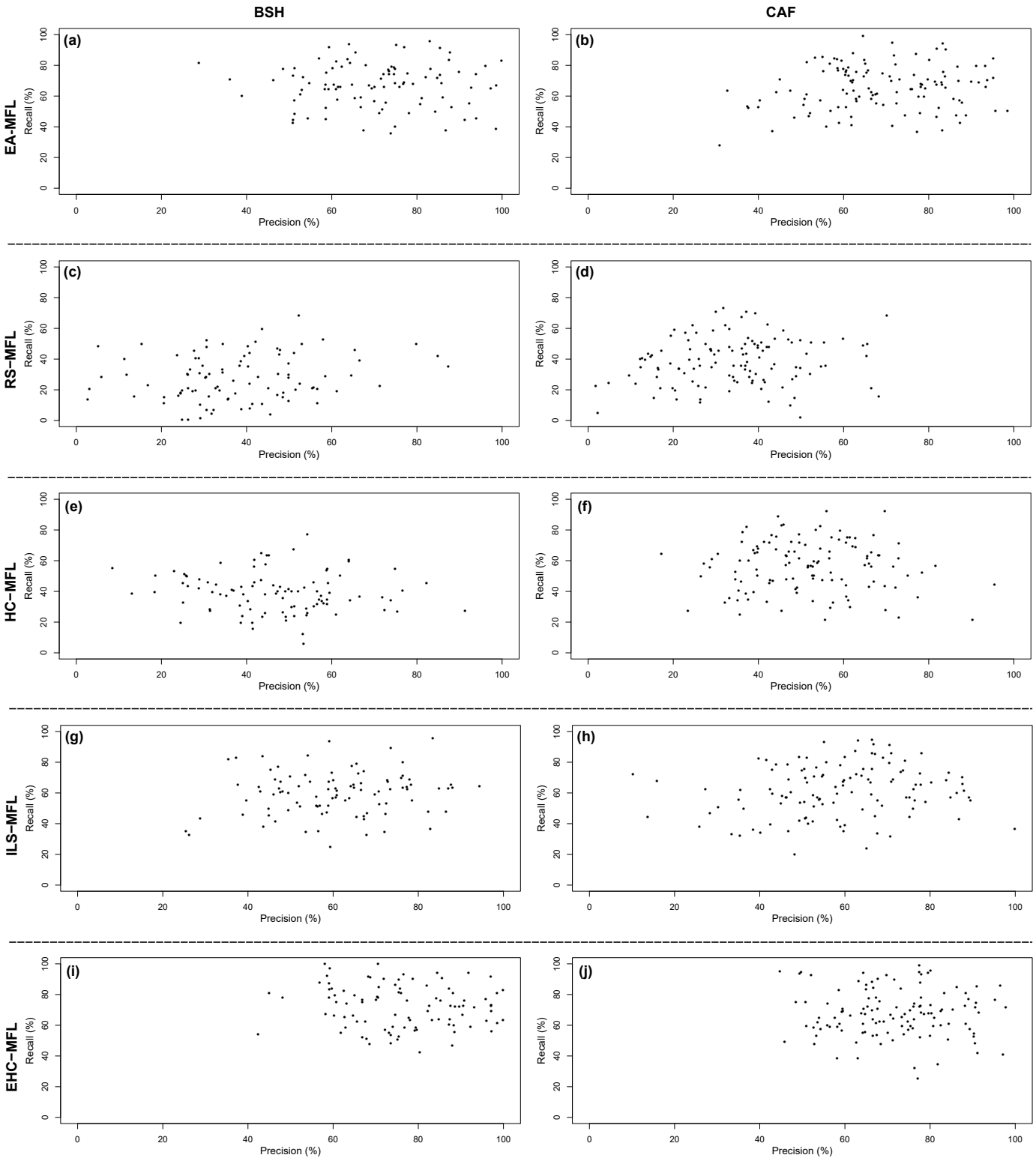


Fig. 8. Mean Precision, Recall, and the F-measure for the two case studies and the five algorithms

EHC-MFL. Then, ILS-MFL is the third option, with values around 20% lower than EHC-MFL. Finally, HC-MFL and RS-MFL are the fourth and fifth options, with values between 25% and 50% lower than the best option.

In response to RQ1, the search algorithms driven by LSA and applied to models from our industrial partners' scenarios have been capable of locating features, giving precision values of up to 76.47% and recall values of up to 72.41%.

1) Statistical significance: The $p - Values$ and statistics of this test are shown in Table III. Since the $p - Values$ shown in this table are smaller than $2x10^{-16}$ in all cases, we reject the null hypothesis. Consequently, we can state that there

TABLE III
QUADE TEST STATISTIC AND $p - Values$

	BSH				CAF			
	Precision	Recall	F-measure	MCC	Precision	Recall	F-measure	MCC
$p - Value$	$\ll 2x10^{-16}$	$\ll 2x10^{-16}$	$\ll 2x10^{-16}$	$\ll 2x10^{-16}$	$\ll 2x10^{-16}$	$\ll 2x10^{-16}$	$\ll 2x10^{-16}$	$\ll 2x10^{-16}$
Statistic	72.319	100.27	132.15	84.132	70.181	50.389	99.646	57.504

TABLE IV
HOLM'S POST HOC $p - Values$

	BSH				CAF			
	Precision	Recall	F-measure	MCC	Precision	Recall	F-measure	MCC
EA vs HC	$1.0x10^{-14}$	$\ll 2x10^{-16}$	$\ll 2x10^{-16}$	$\ll 2x10^{-16}$	$2.7x10^{-12}$	$4.9x10^{-5}$	$1.2x10^{-15}$	$1.3x10^{-10}$
EA vs RS	$\ll 2x10^{-16}$	$\ll 2x10^{-16}$	$\ll 2x10^{-16}$	$\ll 2x10^{-16}$	$\ll 2x10^{-16}$	$\ll 2x10^{-16}$	$\ll 2x10^{-16}$	$\ll 2x10^{-16}$
EA vs ILS	0.0056	0.0017	$4.2x10^{-5}$	0.00014	$7.7x10^{-5}$	0.3453	$9.9x10^{-5}$	$6.0x10^{-5}$
EA vs EHC	0.0056	0.1274	0.004	0.00975	0.466	0.3453	0.11	0.060
HC vs RS	0.0020	$1.3x10^{-5}$	$4.2x10^{-5}$	0.00975	$3.9x10^{-9}$	$1.2x10^{-10}$	$4.9x10^{-12}$	$1.7x10^{-5}$
HC vs ILS	$7.7x10^{-7}$	$3.9x10^{-11}$	$6.6x10^{-13}$	$9.5x10^{-8}$	0.003	0.0081	$3.5x10^{-5}$	0.023
HC vs EHC	$\ll 2x10^{-16}$	$\ll 2x10^{-16}$	$\ll 2x10^{-16}$	$\ll 2x10^{-16}$	$1.9x10^{-14}$	$1.9x10^{-7}$	$\ll 2x10^{-16}$	$3.2x10^{-16}$
RS vs ILS	$5.3x10^{-16}$	$\ll 2x10^{-16}$	$\ll 2x10^{-16}$	$2.4x10^{-15}$	$\ll 2x10^{-16}$	$\ll 2x10^{-16}$	$\ll 2x10^{-16}$	$1.7x10^{-11}$
RS vs EHC	$\ll 2x10^{-16}$	$\ll 2x10^{-16}$	$\ll 2x10^{-16}$	$\ll 2x10^{-16}$	$\ll 2x10^{-16}$	$\ll 2x10^{-16}$	$\ll 2x10^{-16}$	$\ll 2x10^{-16}$
ILS vs EHC	$2.2x10^{-8}$	$5.8x10^{-6}$	$6.9x10^{-12}$	$8.0x10^{-11}$	$3.6x10^{-6}$	0.0359	$7.6x10^{-8}$	$6.3x10^{-9}$

TABLE V
$\hat{A}_{12}$ STATISTIC FOR EACH PAIR OF ALGORITHMS

	BSH				CAF			
	Precision	Recall	F-measure	MCC	Precision	Recall	F-measure	MCC
EA vs HC	0.867241	0.917806	0.962674	0.894640	0.791408	0.657674	0.812649	0.751383
EA vs RS	0.92551	0.965061	0.984809	0.886393	0.947067	0.898094	0.971553	0.898777
EA vs ILS	0.676595	0.655653	0.701931	0.685981	0.668602	0.568540	0.655727	0.637457
EA vs EHC	0.388129	0.410862	0.352431	0.336914	0.446418	0.464244	0.446896	0.447715
HC vs RS	0.663466	0.713487	0.743001	0.601129	0.811591	0.780445	0.864832	0.754457
HC vs ILS	0.254178	0.142687	0.111599	0.217665	0.366949	0.414692	0.348064	0.404003
HC vs EHC	0.072103	0.043240	0.009657	0.034722	0.151083	0.313401	0.140291	0.192268
RS vs ILS	0.153103	0.064290	0.050456	0.197917	0.130524	0.159996	0.095144	0.219589
RS vs EHC	0.041829	0.012967	0.003147	0.069119	0.027013	0.086060	0.014343	0.057646
ILS vs EHC	0.222277	0.272461	0.166721	0.168945	0.272625	0.400792	0.291374	0.308790

are differences in the algorithms for all of the performance indicators evaluated.

Table IV shows the $p-Values$ of Holm's post hoc analysis for each pair of algorithms, case study, and performance indicator. The majority of the $p-Values$ shown in this table are smaller than their corresponding significance threshold value (0.05), indicating that the differences of performance between those algorithms are significant. However, when comparing EA-MFL and EHC-MFL (fourth row), the values for some performance indicators are greater than the threshold, indicating that the differences between those algorithms could be due to the stochastic nature of the algorithms and are not significant.

2) Effect size: Table V shows the values of the effect size statistics. In general, the largest differences were obtained between the EHC-MFL and RS-MFL algorithms (where EHC-MFL achieves better precision than RS-MFL 95% of the times, better recall 98% of the times, better F-measure 99% of the times and better MCC 93% of the times). When comparing EA-MFL and EHC-MFL, the differences are not so big, and the EHC-MFL outperforms EA-MFL around 55% of the times.

In response to RQ2, the EHC-FML algorithm obtained the best performance results among the five algorithms evaluated (see Table II). The statistical analysis performed indicated that EHC-FML will outperform the rest of the algorithms in terms of the metrics analyzed (around 60% of the times when compared to EA-MFL, 78% of the times when compared to ILS, 92% of the times when compared to HC-MFL, and almost all of the times when compared to RS-MFL).

D. Threats to validity

In this section, we present some of the threats to validity. We follow the guidelines suggested by De Oliveira et. al [39] to identify those are applicable to this work.

Conclusion validity threats. We identify four threats of validity of this type. The first threat is the *not accounting for random variation*. To address this threat, we considered 30 independent runs for each feature with each algorithm. The second threat is the *lack of meaningful comparison baseline*. Because we used random search as a standard comparison baseline, this threat is addressed. The third threat is the *lack of formal hypothesis and statistical tests*. In this paper, we employed standard statistical analysis following accepted guidelines [32] to avoid this threat. The fourth threat is the *lack of good descriptive analysis*. In this work, we have used precision, recall, the F-measure and the MCC metrics to

analyze the confusion matrix obtained from the experiments; however, other metrics could be applied. In addition, some works argue that the use of the Vargha and Delaney A12 metrics can be miss-representative [40] and that the data should be pre-transformed before applying them. We did not find any use case for data pre-transformation that applies to our case study.

Internal validity threats. We identify two threats of validity of this type. The first threat is *the poor parameter settings*. 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 search-based techniques in the context of testing. These values have been tested in similar algorithms for feature location [41]. In addition, the choice of the k value in the application of SVD can produce sub-optimal accuracy when using LSA for software artifacts [42]. Nevertheless, we plan to evaluate all of the parameters of our algorithms in a future work. The second threat is the *lack of real problem instances*. The evaluation of this paper was applied to two industrial case studies from two of our partners, BSH and CAF.

Construct validity threats. We identify one threat of validity of this type. The threat is the *lack of assessing the validity of cost measures*. To address this threat, we performed a fair comparison among the algorithms by generating the same number of model fragments and using the same number of fitness evaluations.

External validity threats. We identify two threats of validity of this type. The first threat is the *lack of a clear object selection strategy*, and the second threat is the *lack of evaluations for instances of growing size and complexity*. Both threats are addressed by using two industrial case studies from two of our partners, BSH and CAF. Our instances are collected from real-world problems. In addition, we have two different domains (induction hobs and trains) with different sizes and complexity.

VIII. Related Work

Some works report their industrial experiences transforming legacy products into Product Line assets in a wide range of fields [43], [44], [45]. They focus on capturing guidelines and techniques for manual transformations. In contrast, our approach performs search-based software engineering while taking advantage of the knowledge of the domain experts.

Some works focus on the location of features over models by comparing the models with each other to formalize the variability among them in the form of a Software Product Line:

Wille et al. [46] present an approach where the similarity between models is measured following an exchangeable metric, taking into account different attributes of the models. Then the approach is further refined [47] to reduce the number of comparisons needed to mine the family model.

The authors in [48] propose a generic approach to automatically compare products and locate the feature realizations in terms of a CVL model. In [49], the approach is refined to automatically formalize the feature realizations of new product models that are added to the system. A similar approach is proposed in [50] where the feature location results are validated in an industrial environment.

Martinez et al. [51] propose an extensible approach that is based on comparisons to extract the feature formalization over a family of models. In addition, they provide the means to extend the approach to locate features over any kind of asset based on comparisons. The MoVaPL approach [52] considers the identification of variability and commonality in model variants as well as the extraction of a Model-based Software Product Line (MSPL) from the features identified in these variants. MoVaPL builds on a generic representation of models, making it suitable for any MOF-based models.

However, all of these approaches are based on mechanical comparisons among the models, classifying the elements based on their similarity and identifying the dissimilar elements as the features realizations. In contrast, our work does not rely on model similarity to locate the features; it relies on comparisons with a textual description of the target feature. Specifically, humans are involved in the search by means of the fitness function. Domain experts and application engineers become part of the process, contributing their knowledge of the domain in order to tailor the approach with the feature description. Model fragments that are obtained mechanically are less recognizable by software engineers than those obtained with the participation of software engineers [6].

Lopez-Herrejon et al. [41] evaluate three standard search-based techniques with three objective functions in order to calculate the relationships of a feature model. Their results are slightly better for hill climbing than for the evolutionary algorithm, but they are not statistically significant when the first two objective functions are applied. The authors do not address how each feature is materialized. Our work focuses on the extraction of model fragments that correspond to a feature. Therefore, both works are complimentary: [41] calculates the feature relationships of the feature specification layer, while our work locates the model fragments of the product realization layer (see Fig. 1).

Harman et al. [53] performed a survey on the topic of search-based software engineering applied to SPLs. They present an overview of recent articles that are classified according to themes such as configuration, testing, or architectural improvement. Lopez-Herrejon et al. [54] performed a preliminary systematic mapping study at the connection of search-based software engineering and SPL. They categorized the articles using a known framework for SPL development. These two surveys indicate that search-based software engineering is being applied to SPLs. However, these surveys do not identify works that focus on finding model fragments that materialize the features of the SPL, as our work does.

Font et al. [6] propose a generic approach to locate features among a family of product models based on a human-in-the-loop process. The features are located by the comparison of models and the interaction of engineers that provide their knowledge of the domain. The approach is further refined in [55] and generalized through the use of a genetic algorithm to create the model fragments. They introduce a genetic operator for mutation that can work with a single model fragment and a crossover operator that combines two different product models.

The results show that the use of a genetic algorithm allows the approach to provide accurate location of features in spite of inaccurate information on the part of the user.

However, since the work in [55] is designed to locate features by comparisons among the members of a family, the participation of the software engineers is limited and the resultant model fragments are less recognizable to them. In contrast, in this paper, we present five algorithms (EA-MFL, HC-MFL, RS-MFL, ILS-MFL, and EHC-MFL), which are addressed by a feature description given in natural language. Our fitness function makes use of LSA to measure the similarity with the description provided and to store the model fragments.

IX. Conclusions

This work proposes and compares five search algorithms to locate features over a family of models (MFL): Evolutionary Algorithm (EA-MFL), Random Search (RS-MFL) used as a sanity check, steepest Hill Climbing (HC-MFL), Iterated Local Search with random restarts (ILS-MFL), and a hybrid between Evolutionary and Hill Climbing (EHC-MFL). We apply Latent Semantic Analysis (LSA) as the fitness function.

In this work, we address two research questions: (RQ1) Can SBSE techniques driven by LSA be applied to locate features in product models from real industrial scenarios?; (RQ2) If so, which evolutionary algorithm produces the best results in terms of solution quality? To do so, we conducted an evaluation in BSH (the manufacturer of home appliances) and in CAF (the manufacturer of rolling stock). We report our evaluation, including the experimental setup, the results, the statistical analysis, and the threats to validity identified.

The results show that SBSE techniques can be applied to locate features in product models. Specifically, the use of genetic operations for models in combination with the EHC-MFL algorithm provided the best results in our study. This demonstrates that SBSE for feature location at the model level can be applied in real world environments.

Acknowledgments

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). The authors are very grateful to the anonymous reviewers for their valuable suggestions and comments to improve the quality of this paper.

References

[1] M. Harman, "Why the virtual nature of software makes it ideal for search based optimization," in *Proceedings of the 13th International Conference on Fundamental Approaches to Software Engineering*, ser. FASE'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 1–12.

[2] W. B. Langdon and M. Harman, "Optimizing existing software with genetic programming," *IEEE Transactions on Evolutionary Computation*, vol. 19, no. 1, pp. 118–135, Feb 2015.

[3] B. Dit, M. Revelle, M. Gethers, and 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 and M. Chechik, "A survey of feature location techniques," in *Domain Engineering*, I. Reinhartz-Berger, A. Sturm, T. Clark, S. Cohen, and J. Bettin, Eds. Springer Berlin Heidelberg, 2013, pp. 29–58.

[5] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of Software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2013.

[6] J. Font, L. Arcega, Ø. Haugen, and 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.

[7] T. K. Landauer, P. W. Foltz, and D. Laham, "An introduction to latent semantic analysis," *Discourse processes*, vol. 25, no. 2-3, pp. 259–284, 1998.

[8] C. D. Manning, P. Raghavan, H. Schütze *et al.*, *Introduction to information retrieval*. Cambridge university press, 2008, vol. 1, no. 1.

[9] M. Shepperd and S. MacDonell, "Evaluating prediction systems in software project estimation," *Information and Software Technology*, vol. 54, no. 8, pp. 820 – 827, 2012, special Issue: Voice of the Editorial BoardSpecial Issue: Voice of the Editorial Board.

[10] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Softw. Test. Verif. Reliab.*, vol. 24, no. 3, pp. 219–250, May 2014. [Online]. Available: http://dx.doi.org/10.1002/stvr.1486

[11] Ø. Haugen, B. Moller-Pedersen, J. Oldevik, G. Olsen, and A. Svendsen, "Adding standardized variability to domain specific languages," in *Software Product Line Conference, 2008. SPLC '08. 12th International*, Sept 2008, pp. 139–148.

[12] "Meta object facility (mof) version 2.4.1," 2013, object Management Group (OMG) Specification.

[13] M. M. Lehman, J. Ramil, and G. Kahen, *A paradigm for the behavioural modelling of software processes using system dynamics*. Citeseer, 2001.

[14] M. Gethers, R. Oliveto, D. Poshyvanyk, and A. D. Lucia, "On integrating orthogonal information retrieval methods to improve traceability recovery," in *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, Sept 2011, pp. 133–142.

[15] A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella, "Labeling source code with information retrieval methods: an empirical study," *Empirical Software Engineering*, vol. 19, no. 5, pp. 1383–1420, 2014.

[16] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 837–847.

[17] S. Gupta, S. Malik, L. Pollock, and K. Vijay-Shanker, "Part-of-speech tagging of program identifiers for improved text-based software engineering tools," in *2013 21st International Conference on Program Comprehension (ICPC)*, May 2013, pp. 3–12.

[18] G. Capobianco, A. D. Lucia, R. Oliveto, A. Panichella, and S. Panichella, "On the role of the nouns in ir-based traceability recovery," in *Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on*, May 2009, pp. 148–157.

[19] ——, "Improving ir-based traceability recovery via noun-based indexing of software artifacts," *Journal of Software: Evolution and Process*, vol. 25, no. 7, pp. 743–762, 2013.

[20] S. Zamani, S. P. Lee, R. Shokripour, and J. Anvik, "A noun-based approach to feature location using time-aware term-weighting," *Information and Software Technology*, vol. 56, no. 8, pp. 991 – 1011, 2014.

[21] A. Kuhn, S. Ducasse, and T. Gírba, "Semantic clustering: Identifying topics in source code," *Inf. Softw. Technol.*, vol. 49, no. 3, pp. 230–243, Mar. 2007.

[22] P. van der Spek, S. Klusener, and P. van de Laar, *Complementing Software Documentation*. Dordrecht: Springer Netherlands, 2011, pp. 37–51.

[23] M. Affenzeller, S. Winkler, S. Wagner, and A. Beham, *Genetic Algorithms and Genetic Programming: Modern Concepts and Practical Applications*, 1st ed. Chapman & Hall/CRC, 2009.

[24] S. Luke, *Essentials of Metaheuristics*, 2nd ed. Lulu, 2013, available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/.

[25] J. Baxter, "Local optima avoidance in depot location," *The Journal of the Operational Research Society*, vol. 32, no. 9, pp. 815–819, 1981.

[26] H. R. Lourenço, O. C. Martin, and T. Stützle, *Iterated Local Search*. Boston, MA: Springer US, 2003, pp. 320–353.

[27] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework 2.0*, 2nd ed. Addison-Wesley Professional, 2009.

[28] D. Dyer, "The watchmaker framework for evolutionary computation," http://watchmaker.uncommons.org/, 2016, [Online; accessed 7-April-2016].

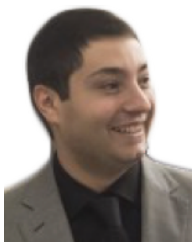
[29] "Apache opennlp: Toolkit for the processing of natural language text," https://opennlp.apache.org/, 2016, [Online; accessed 7-April-2016].

[30] "The english (porter2) stemming algorithm," http://snowball.tartarus.org/algorithms/english/stemmer.html, 2016, [Online; accessed 7-April-2016].
[31] "Efficient java matrix library," http://ejml.org/, [Online; accessed 7-April-2016].
[32] A. Arcuri and G. Fraser, "Parameter tuning or default values? an empirical investigation in search-based software engineering," *Empirical Software Engineering*, vol. 18, no. 3, pp. 594–623, 2013.
[33] A. Kotelyanskii and G. M. Kapfhammer, "Parameter tuning for search-based test-data generation revisited: Support for previous results," in *2014 14th International Conference on Quality Software*, Oct 2014, pp. 79–84.
[34] A. S. Sayyad, J. Ingram, T. Menzies, and H. Ammar, "Scalable product line configuration: A straw to break the camel's back," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, Nov 2013, pp. 465–474.
[35] S. García, A. Fernández, J. Luengo, and F. Herrera, "Advanced non-parametric tests for multiple comparisons in the design of experiments in computational intelligence and data mining: Experimental analysis of power," *Inf. Sci.*, vol. 180, no. 10, pp. 2044–2064, May 2010.
[36] W. J. Conover, *Practical Nonparametric Statistics, 3rd Edition*. Wiley, 1999.
[37] A. Vargha and 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*, vol. 25, no. 2, pp. 101–132, 2000.
[38] R. J. Grissom and J. J. Kim, *"Effect sizes for research: A broad practical approach*. Mahwah, NJ: Earlbaum, 2005.
[39] M. de Oliveira Barros and A. C. Dias-Neto, "0006/2011-threats to validity in search-based software engineering empirical studies," *RelaTe-DIA*, vol. 5, no. 1, 2011.
[40] G. Neumann, M. Harman, and S. Poulding, *Transformed Vargha-Delaney Effect Size*. Cham: Springer International Publishing, 2015, pp. 318–324. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-22183-0_29
[41] R. E. Lopez-Herrejon, L. Linsbauer, J. A. Galindo, J. A. Parejo, D. Benavides, S. Segura, and A. Egyed, "An assessment of search-based techniques for reverse engineering feature models," *Journal of Systems and Software*, vol. 103, pp. 353 – 369, 2015.
[42] A. Panichella, B. Dit, R. Oliveto, M. D. Penta, D. Poshyvanyk, and A. D. Lucia, "Parameterizing and assembling ir-based solutions for se tasks using genetic algorithms," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, March 2016, pp. 314–325.
[43] K. Kim, H. Kim, and W. Kim, "Building software product line from the legacy systems "experience in the digital audio and video domain"," in *Software Product Line Conference, 2007. SPLC 2007. 11th International*, Sept 2007, pp. 171–180.
[44] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi, "Refactoring a legacy component for reuse in a software product line: A case study: Practice articles," *J. Softw. Maint. Evol.*, vol. 18, no. 2, pp. 109–132, Mar. 2006.
[45] H. Lee, H. Choi, K. Kang, D. Kim, and Z. Lee, "Experience report on using a domain model-based extractive approach to software product line asset development," in *Formal Foundations of Reuse and Domain Engineering*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, vol. 5791, pp. 137–149.
[46] D. Wille, S. Holthusen, S. Schulze, and 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.
[47] S. Holthusen, D. Wille, C. Legat, S. Beddig, I. Schaefer, and 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.
[48] X. Zhang, Ø. Haugen, and 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.
[49] X. Zhang, Ø. Haugen, and B. Møller-Pedersen, "Augmenting product lines," in *Software Engineering Conference (APSEC), 2012 19th Asia-Pacific*, vol. 1, Dec 2012, pp. 766–771.
[50] J. Font, M. Ballarín, Ø. Haugen, and C. Cetina, "Automating the variability formalization of a model family by means of common variability language," in *Proceedings of the 19th International Conference on Software Product Line (SPLC)*, 2015, pp. 411–418.
[51] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and 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.
[52] J. Martinez, T. Ziadi, T. F. Bissyand, J. Klein, and Y. l. Traon, "Automating the extraction of model-based software product lines from model variants (t)," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, Nov 2015, pp. 396–406.
[53] M. Harman, Y. Jia, J. Krinke, W. B. Langdon, J. Petke, and Y. Zhang, "Search based software engineering for software product line engineering: A survey and directions for future work," in *Proceedings of the 18th International Software Product Line Conference - Volume 1*, ser. SPLC '14. New York, NY, USA: ACM, 2014, pp. 5–18.
[54] R. E. Lopez-Herrejon, J. Ferrer, F. Chicano, L. Linsbauer, A. Egyed, and E. Alba, "A hitchhiker's guide to search-based software engineering for software product lines," *CoRR*, vol. abs/1406.2823, 2014.
[55] J. Font, L. Arcega, Ø. Haugen, and C. Cetina, "Feature location in model-based software product lines through a genetic algorithm," in *15th International Conference on Software Reuse*, ser. ICSR 2016, Limassol, Cyprus, Jun 2016.

Jaime Font is a PhD student in computer science at the University of Oslo and a researcher in the SVIT Research Group at San Jorge University. His research interests include reverse engineering, evoluationary computation, and variability modeling. He received an MSc in Advanced Software Technologies from San Jorge University. Contact him at jfont@usj.es

Lorena Arcega is a PhD student in computer science at the University of Oslo and a researcher in the SVIT Research Group at San Jorge University. Her research interests are software evolution, variability modeling, and models at run-time. She received an MSc in Advanced Software Technologies from San Jorge University. Contact her at larcega@usj.es

Øystein Haugen is a professor at Østfold University College and senior researcher at SINTEF ICT. His research focuses on cyber-physical systems, variability modeling and software product lines, object oriented languages and methods, software engineering, and practical formal methods. Contact him at oystein.haugen@hiof.no



Carlos Cetina is an associate professor at San Jorge University and the head of the SVIT Research Group. His research focuses on software product lines, variability modeling, and model-driven development. Cetina received a PhD in computer science from the Universitat Politècnica de València. Contact him at ccetina@usj.es

APPENDIX
ALGORITHMS PSEUDOCODES AND PARAMETERS

TABLE VI
PARAMETERS FOR THE ALGORITHMS

Parameter	Description	Value
Size	Size of the population	100
StopCondition	Budget allocated to run the algorithm (s)	80
μ	Number of parents for the crossover	2
λ	Number of offspring from μ parents	1
p_c	Probability of crossover	0.75
p_m	Probability of mutation, where n is the length of the chromosome being mutated	1/n

Algorithm 1 Evolutionary Algorithm

$P \leftarrow initPopulation(inputData, size)$
while (!StopCondition) **do**
  $Bestevaluatefitness(P)$
  $P \leftarrow breedPopulation(P)$
end while
return $Best$

function initPopulation(inputData,size)
  $P \leftarrow []$ ▷ Initial population empty
  for $i = 1$ to $size$ **do**
    $F \leftarrow randomFragment(inputData)$
    $P \leftarrow P + F$ ▷ Add the new individual
  end for
  return P
end function

function breedPopulation(P,size)
  $P_0 \leftarrow []$ ▷ empty population
  for $i = 1$ *to* $size$ **do**
    $parents \leftarrow selectionParents(P)$
    $offspring \leftarrow crossover(parents, p_c)$
    $offspring \leftarrow mutation(offspring, p_m)$
    $P_0 \leftarrow P_0 + offspring$ ▷ Add the new offspring
  end for
  return P_0
end function

Algorithm 2 Random Search

$Best \leftarrow randomFragment(inputData)$
$I \leftarrow 0$
while (!$StopCondition$)) **do**
  $S \leftarrow randomFragment(inputData)$
  if $(fitness(S) > fitness(Best))$ **then**
    $Best \leftarrow S$
  end if
  $I \leftarrow I + 1$
end while
return $Best$

Algorithm 3 Steepest Ascent Hill Climbing with Replacement

$S \leftarrow randomFragment(inputData)$
$NSize \leftarrow$ number of neighbors desired
$I \leftarrow 0$
$Best \leftarrow S$
while (!$StopCondition$) **do**
  $X \leftarrow 0$
  while $X < NSize$ **do**
    $S' \leftarrow tweak(Best)$
    if $(fitness(S') > fitness(S))$ **then**
      $S \leftarrow S'$
    end if
    $X \leftarrow X + 1$
  end while
  if $(fitness(S) > fitness(Best))$ **then**
    $Best \leftarrow S$
  end if
  $I \leftarrow I + 1$
end while
return $Best$

Algorithm 4 Iterated Local Search (ILS) with Random Restarts

$Current \leftarrow randomFragment(inputData)$
$Times \leftarrow$ distribution of time intervals
$Home \leftarrow Current$
$Best \leftarrow Current$
while (!$StopCondition$) **do**
  $time \leftarrow$ random time chosen from $Times$
  while (!$StopCondition$ && $time \neq 0$) **do**
    $Aux \leftarrow tweak(Current)$
    if $(fitness(Aux) > fitness(Current)$ **then**
      $Current \leftarrow Aux$
    end if
  end while
  if $(fitness(Current) > fitness(Best)$ **then**
    $Best \leftarrow Current$
  end if
  $Home \leftarrow newHomeBase(Home, Current)$
  $Current \leftarrow perturb(Home)$
end while
return $Best$

Algorithm 5 Hybrid between Evolutionary and Hill-Climbing

$HCIter \leftarrow$ number of iterations to Hill-Climb
$P \leftarrow initPopulation(InputDataSize)$
while (!$StopCondition$) **do**
  $fitness(P)$
  for P_i *in* P **do**
    $P_i \leftarrow Hill - Climb(P_i)forHCIter$
    if $(fitness(P_i) > fitness(Best))$ **then**
      $Best \leftarrow P_i$
    end if
  end for
  $P \leftarrow breedPopulation(P)$
end while
return $Best$