Chapter 1
Feature Location in Models (FLiM): Design time and Runtime

Lorena Arcega, Jaime Font, Øystein Haugen and Carlos Cetina

Abstract In this chapter, we apply feature location to automate the identification and extraction of the features existing among a family of product models and re-engineering them into a model-based SPL. To address the feature location in software models (FLiM) challenge, we present two approaches: at design time (FLiMEA) and at runtime (FLiMRT). Both FLiMEA and FLiMRT approaches are different but complementary. FLiMEA takes information from design time models while FLiMRT takes information from runtime models. The FLiMEA approach combines Genetic Operations and Information Retrieval. Given a model and a description of a possible feature, model fragments extracted from the model are generated using genetic operation and are assessed using an information retrieval technique to rank the candidates based on the similarity with the feature description. The FLiMRT approach leverages the use of software architecture models at runtime. The information is collected in the software architecture model at runtime and each model element is assessed based on its similarity to the feature description. We evaluated our approaches in two real-world industrial case studies: BSH and CAF. The application of FLiMEA shows that the values of recall and precision are higher than 85 per cent while FLiMRT ranks the relevant elements in the top ten positions of the ranking in 84 per cent of the cases.

Lorena Arcega, Jaime Font and Carlos Cetina
Universidad San Jorge, SVIT Research Group, Autova A-23 km. 299, 50830 Zaragoza, Spain, e-mail: {larcega,jfont,ccetina}@usj.es

Lorena Arcega
University of Oslo, Department of Informatics, Postbooks 1080 Blindern, 0316 Oslo, Norway

Øystein Haugen
Østfold University College, Faculty of Computer Science, Postboks 700, 1757 Halden, Norway e-mail: oystein.haugen@hiof.no

1.1 Introduction

Software Product Lines (SPLs) aim at reducing development cost and time to market while improving quality of software systems by exploiting commonalities and managing variabilities across a set of software applications [52]. The SPL engineering paradigm separates two processes; domain engineering (where the commonalities are identified and realized as reusable assets) and application engineering (where specific software products are derived by reusing the variability of the SPL) [12]. Traditionally, a domain analysis is performed to build a feature model that captures the variability of the system in terms of features [13, 29]. The domain knowledge from the experts is captured and used to build the library of reusable assets.

A recent survey [6] reveals that most of the SPLs are built when there are already products; therefore, the set of existing products is re-engineered into an SPL [34]. This is known as the extractive approach to SPLs [34]; it capitalizes on existing systems to initiate a product line, formalizing variability among a set of similar products into a variability model. The resulting SPL is capable of generating the products used as input (among others) with the benefit of having the variability among the products formalized, enabling a systematic reuse.

Feature Location is known as the process of finding the set of software artifacts that realize a particular feature. This topic has gained momentum during recent years [16, 57]. However, most of the research efforts in feature location have been directed towards the location of features into source code artifacts, neglecting other software artifacts such as models [43, 44, 67]. In addition, 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 [25].

The task of feature location in models may appear easy, however it becomes very complex in the models of our industrial partners [51]. For example, suppose we ask the domain experts of one of our industrial partners, CAF in this case, to manually locate the model elements that correspond to the 121 features of the data set provided. Taking into account that the data set comprises 23 trains and the model of each train has more than 1200 model elements, at least 27,600 model elements should be evaluated. To assess a model element, it is reasonable to consider its properties. In the data set, each element has about 15 properties. Therefore, about 414,000 properties of model elements should be considered. Assuming that a domain expert only needs 1 second to consider a property of a model element, the domain expert needs 4.79 days to manually locate each feature. Considering the 121 features and 19 domain experts, the result is 30.17 years.

Domain experts could make use of simple text search tools in the models, but these tools would not prevent domain experts from first knowing the models of the trains. There is no domain expert who knows all models completely in CAF. Although we ignore that domain experts can forget models

that belong to trains manufactured over two decades as is the case in CAF, the models have always been created by different domain experts. Moreover, the models may have been maintained by other domain experts who have not participated in the creation. Time improvements because of the learning effect, or locating several features simultaneously are not accounted here, but these improvements could also be source of errors which take time to fix.

In addition, the 30.17 years do not include the time that is necessary to reach a consensus on 19 solutions for each of the 121 features. In an industrial environment like in CAF, the domain experts are distributed in three different cities of Spain (Zaragoza, Beasain and Bizkaia). This geographical distribution implies that the domain experts are not used to carrying out consensus tasks, which can negatively influence the time they need to agree on the solutions.

Therefore, feature location in real-world models is not a trivial task. From the 121 features of the data set, only the model elements of 43 features are documented in CAF. The documentation of these 43 features is the result of months of manual work with external consultants to address certification needs or bugs. Moreover, this data set is made up of tramway models, but the need to locate features is also present in more CAF models of similar complexity as subway models, or in more complex models such as suburban and high speed.

We can apply feature location to automate the identification and extraction of the features existing among a family of product models and reengineering them into a model-based SPL (an SPL whose final products are models) by establishing precisely the variability between the features. To address this challenge we present an approach that turns a set of similar but different product models with no variability specification into a set of product models with a formal variability definition that specifies the commonalities and variability among them [23].

In addition, we can apply feature location in systems that work with models at runtime. We use the information extracted from software architecture models at runtime to perform feature location. In our dynamic feature location approach [3], the software engineer executes a scenario, which uses the desired feature to be located. The information is collected in the software architecture model at runtime. Then, our approach filters the trace in order to extract the relevant elements of the software models.

In the evaluation, we have applied our approaches to two industrial domains: BSH and CAF. BSH is one of the largest manufacturers of home appliances in Europe. Its induction division has been producing induction hobs under the brands of Bosch and Siemens for the last 15 years. CAF produces a family of PLC software to control the trains they manufacture, which has been under development for more than 25 years. The firmware for their products is generated from models using a model-based approach. The results of the evaluations show a good performance of our approaches. The application of FLiMEA shows that the mean values of recall and precision

are around 72.99% for BSH and 68.34% for CAF. Our FLiMRT approach ranks the relevant elements in the top ten positions of the ranking in 84% of the cases.

1.2 Background

This section provides some basic background for understanding this chapter. Specifically, we present Software Product Lines (SPLs), feature location, information retrieval, and evolutionary algorithms. In addition, the Domain-Specific Language (DSL) used by our industrial partner to formalize their products, the IHDSL. It will be used through the rest of the chapter to present a running example. Then, the Common Variability Language (CVL) is presented, CVL is the language used by our approach to formalize the model fragments.

1.2.1 Software Product Lines

Mass production was popularized by Henry Ford in the early 20th Century. McIlroy coined the term software mass production in 1968 [45]. It was the beginning of SPLs. In 1976, Parnas introduced the notion of software program families as a result of mass production [50]. The use of features (to drive mass production) was proposed by Kang in the early 1990s [29]. Shortly, the first conferences appeared turning SPL into a new body of research [37].

SPLs are defined as "a set of software-intensive systems, sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" [12, 17]. This definition can be redefined into five major issues:

1. Products. SPL shift the focus from single software system development to SPL development. The development processes are not intended to build one application, but a number of them (e.g., 10, 100, 10,000, or more). This forces a change in the engineering processes where a distinction between domain engineering and application engineering is introduced. Doing so, the construction of the reusable assets (platform) and their variability is separated from production of the product-line applications.
2. Features. Features are units (i.e., increments in application functionality) by which different products can be distinguished and defined within an SPL [5].
3. Domain. An SPL is created within the scope of a domain. A domain is a specialized body of knowledge, an area of expertise, or a collection of related functionality [48].

4. Core Assets. A core asset is an artifact or resource that is used in the production of more than one product in an SPL [12].
5. Production Plan. It states how each product is produced. The production plan is a description of how core assets are to be used to develop a product in a product line and specifies how to use the production plan to build the end product [9]. The production plan ties together all the reusable assets to assemble (and build) end products. Synthesis is a part of the production plan.

SPLs (or system families) provide a highly successful approach to strategic reuse of assets within an organization. A standard SPL consists of a product line architecture, a set of software components and a set of products. A product consists of a product architecture, derived from the product line architecture, a set of selected and configured product line components and product specific code.

Therefore, SPL engineering is about producing families of similar systems rather than the production of individual systems. SPL engineering consists of three main processes: domain engineering (also called core asset development), application engineering (also called product development) and management. These three processes are complementary and provide feedback to each other.

Domain engineering is, among others, concerned with identifying the commonality and variability for the products in the product line and implementing the shared artifacts such that the commonality can be exploited while preserving the required variability. Using a "design-for-reuse" approach, domain engineering (core asset development [12]) is on charge of determining the commonality and the variability among product family members. In general, domain engineering is divided into domain analysis, domain design and domain implementation.

During application engineering, individual products are developed by selecting and configuring shared artifacts and, where necessary, adding product specific extensions. To achieve this, it reuses the reusable assets developed previously. This process is subdivided into application analysis, application design and application implementation.

Management is responsible for giving resources, coordinating, and supervising domain and application engineering activities. See [12, 52] for more details about the processes.

In SPL processes, variability is made explicit through variation points. A variation point represents a delayed design decision. When the architect or designer decides to delay the design decision, he or she has to design a variation point. The design of the variation point requires several steps: (1) the separation of the stable and variant behaviour, (2) the definition of an interface between these types of behaviour, (3) the design of a variant management mechanism and (4) the implementation of one or more variants. Given a variation point, it can be bound to a particular variant. For each variation point, the set of variants may be open, i.e. more variants can be

added, or closed, i.e. no more variants can be added. Overall, during domain engineering new variation points are introduced, whereas during application engineering these variation points are bound to selected variants

1.2.2 Feature Location

In software systems a feature represents a functionality that is defined by requirements and accessible to developers and users [16]. Identifying an initial location in the software artifacts that corresponds to a specific functionality is known as feature location [7].

The majority of existing approaches for feature location in program code can be divided in three categories: static, dynamic, and hybrid. Static feature location approaches examine structural information such as control or data flow dependencies. These approaches require a set of software artifacts which serve as a starting point for the analysis in order to generate program elements relevant to the initial set. This initial set is usually specified by the software engineer. Dynamic feature location approaches examine a software system's execution. They are often used for feature location when features can be invoked and observed during runtime. Typically, these approaches rely on a post-mortem analysis of an execution trace. The combination of more than one type of analysis results in a hybrid approach for feature location.

Existing approaches for feature location in models rely on mechanical comparisons to find model differences. First, several comparisons among the product models are performed. Then, a set of model fragments is extracted based on the differences and common parts spotted among the models. Identical elements are extracted as common parts of the product line, similar elements are extracted as variable alternative parts, and unmatched elements are extracted as variable optional parts. As a result, the variability existing among the set of similar product models is formalized.

1.2.3 Information Retrieval

Information Retrieval (IR) is the task, given a set of documents and a user query, of finding the relevant documents. There are many information retrieval techniques: program analysis dependencies [11, 19, 20, 33], textual similarity [38, 42, 55, 60], trace analysis [21, 53, 65, 66], type systems [18, 31, 32, 30] or propositional logic [2, 14, 46, 47]. In these approaches, we focus on information retrieval techniques based on textual similarity.

The IR techniques based on textual similarity, are based on mathematical and statistical methods to determine the similarity between different collections of texts. Three of the most popular IR techniques are: Vector Space

Model (VSM) [59], Latent Semantic Analysis (LSA) [36] or Latent Dirichlet Allocation (LDA) [8].

The current results are ambiguous and contradictory about which technique provides the best performance [64]. However, some works state that LSI performs better working with bug reports [10] or with text [54], while VSM works better with source code. This is due to the reason that VSM works very well in case of exact match while LSI retrieves relevant documents based on the semantic similarity.

We decided to apply Latent Semantic Analysis (LSA) to analyze the relationships between the description of the features and the bugs provided by the user and the model. This decision is because of product models are representations at a higher abstraction level than the source code, and the language used to build them is closer to the bug description language; similar to text.

1.2.4 Evolutionary Algorithms

The approach for feature location in models at design time explined is in this chapter comes in the form of an Evolutionary Algorithm. In Evolutionary Algorithms, a population of individuals (candidate solutions for the problem) is evolved and ssessed through several iterations in the search for the best possible individual. When applied to model artifacts, the population of individuals will be in the form of model fragments. These individuals need to be properly encoded, enabling the evolutionary algorithm to work efficiently with them. Next, each candidate solution from the population is evaluated using a fitness function (a formalization of the overall quality goal) to determine how well it performs as a solution to the problem. As a result, the population of solutions is ranked depending on their fitness value and, based on the ranked population, some genetic manipulations are performed over the individuals. This cycle of genetic manipulations and assessment will be repeated until some stop criteria is met.

As we said, each possible solution to the problem (called individual) needs to be encoded so the genetic operations can be applied to them. Traditionally, individuals are encoded as a fixed-size string of binary values, but other encodings can be used such as tree encodings. In fact, it is suggested [15] to use an encoding natural for the problem and then devise genetic operations capable of working for that specific encoding. The individuals of our problem are model fragments (extracted from some product model); therefore, the encoding must be able to represent a model fragment extracted from a product model.

Next, the fitness function is used as an heuristic to guide the search performed by the evolutionary algorithm. To do so, the function assigns a fitness value to each of the model fragments based on their quality as feature real-

ization. This information can be used in two ways: to determine that the algorithm should terminate as a desirable level of fitness has been reached and to determine the best candidates to be used as parents for the next generation.

Finally, different operations are performed to manipulate the individuals, with the hope that manipulated individuals (so-called offspring) will perform better after manipulation. Then, to perform the genetic manipulations some parents are selected based on previous fitness assessment, giving priority to the solutions with higher fitness values. Then two types of genetic operations can be performed: crossover, that combines two parents into a new individual; mutation, the individual is evolved and some of its characteristics are modified (added or removed).

1.2.5 The Induction Hob Domain

Traditionally, stoves have a rectangular shape and feature four rounded areas that become hot when turned on. Therefore, the first Induction Hobs (IHs) created provided similar capabilities. However, the induction hobs domain is constantly evolving and, due to the possibilities provided by the induction phenomena and the electronic components present in the induction hobs, a new generation of IHs has emerged [1].

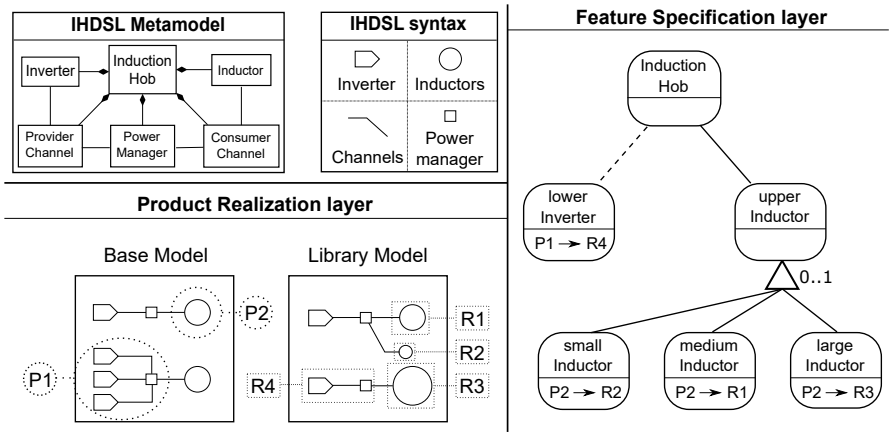

Fig. 1.1 IHDSL and CVL applied to it.

For instance, the newest IHs feature full cooking surfaces, where dynamic heating areas are automatically calculated and activated or deactivated depending on the shape, size, and position of the cookware placed on top. There

[1] freeInduction cooktop demo: https://www.youtube.com/watch?v=EZ8UAvt9paI

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 being possible at the cost of increasing the software complexity. The Domain-Specific Language (DSL) used by our industrial partner to specify the Induction Hobs (IHDSL) is composed of 46 meta-classes, 74 references among them and more than 180 properties. However, in order to gain legibility and due to intellectual property rights concerns, in this section we use a simplified subset of the IHDSL (see the top of Fig. 1.1).

Inverters are in charge of converting the input electric supply to match the specific requirements of the induction hob. Specifically, the amplitude and frequency of the electric supply needs to be precisely modulated in order to improve the efficiency of the IH and to avoid resonance. Then, the energy is transferred to the hotplates 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 runtime. The path followed by the energy through the channels is controlled by the power manager.

Inductors are the elements where energy is transformed into an electromagnetic field. Inductors are composed of a conductor that is usually wound into a coil. However, inductors vary in their shape and size, resulting in different power supply needs in order to achieve performance peaks. Inductors can be organized into groups in order to heat larger cookware while sharing the user interface controllers. Each group of inductors can have different particularities; for instance, some of them can be divided into independent zones or others can grow in size adapting to the size of the cookware being placed on top of them. Some of the groups of inductors are made at design time, while others can occur at runtime (depending on the cookware placed on top).

Bottom left part of Fig. 1.1 depicts an example of a product model specified with the IHDSL. The product model contains four inverters used to power two different inductors. The upper inductor is powered by a single inverter while the lower inductor is powered by the combination of three different inverters. Power managers act as hubs to perform the connection between the inverters and the inductors.

1.2.6 The Common Variability Language applied to Induction Hobs

Our approach formalizes the features located as model fragments, the subset of model elements from a whole model that realizes a particular feature. We use the Common Variability Language (CVL) to formalize the model fragments used as features.

The Common Variability Language (CVL) [26, 49, 63] was recommended for adoption as a standard by the Architectural Board of the Object Management Group and is our industrial partners choice for specifying and resolving variability. CVL defines variants of a base model (conforming to MOF) by replacing variable parts of the base model by alternative model replacements found in a library model.

The variability specification through CVL is divided across two different layers: the feature specification layer (where variability can be specified following a feature model syntax) and the product realization layer (where variability specified in terms of features is linked to the actual models in terms of placements, replacements and substitutions).

The base model is a model described by a given Domain-Specific Language (DSL) which serves as a base for different variants defined over it. In CVL the elements of the base model that are subject to variations are the placement fragments (hereinafter placements). 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 described in the same DSL as the base model that will serve as basis to define alternatives for a placement. Each one of the alternatives for a placement is a replacement fragment (hereinafter replacement). Similarly to placements, a replacement can be any element or set of elements that can be used as variation for a replacement.

CVL defines variants of the base model by means of fragment substitutions. Each substitution references to a placement and a replacement and includes the information necessary to substitute the placement by the replacement. In other words, each placement and replacement is defined along with its boundaries, which indicate what is inside or outside each fragment (placement or replacement) in terms of references among other elements of the model. Then, the substitution is defined with the information of how to link the boundaries of the placement with the boundaries of the replacement. When a substitution is materialized, the base model (with placements substituted by replacements) continues to conform to the same metamodel.

Fig. 1.1 shows an example of variability specification of IH through CVL. In the product realization layer, two placements are defined over an IH base model (P1 and P2). Then, four replacements are defined over an IH library model (R1, R2, R3, and R4). In the feature specification layer, a Feature Model is defined that formalizes the variability among the IH based on the placements and replacements previously defined. For instance, P1 can only be substituted by R4 (which is optional), but P2 can be replaced by R1, R2, or R3. Note that each fragment has a signature, which is a set of references going from and towards that replacement. A placement can only be replaced by replacements that match the signature. For instance, the P2 signature has a reference from a power manager (outside the placement) to an inductor (inside the placement), while the R4 signature is a reference from a power manager (inside the replacement) to an inductor (outside the replacement). P2 cannot be substituted by R4 since their signatures do not match.

1.3 Relation between FLiMEA and FLiMRT

The approaches presented in this chapter need some model elements in order to work. To perform feature location with FLiMRT, we need a system with at least, design time models and runtime models. And to perform feature location with FLiMEA, we need a system with at least design time models. In addition, taking into account the results obtained in our evaluations, we realized that our approaches are complementary. In the real word, before locating a feature, in most cases, we do not know the source of the feature for sure.

We recommend to apply specific approaches taking into account the problem to be solved and the system where the feature is located. If the results are not satisfactory, then we can manually refine the solutions or apply other approaches. For example, if we want to locate a feature and we have a system with models at runtime. Our recommendation is to use the FLiMRT approach as the starting point. If the results obtained by FLiMRT are not useful, other approaches can be launched since they explore a larger solution space than FLiMRT.

1.4 FLiM at Design time (FLiMEA)

In this section we explain our approach for feature location in product models. As we said before, the objective of the approach is to provide a subset of elements from a given product model that realizes a particular feature being requested by the user.Then, we are going to use Genetic Operations to iterate over a population of model fragments and evolve them using genetic operations. The output is a list of model fragments that might realize the feature. This list is assessed and ranked taking into account the information provided by the user as input. The entire explanation of the approach can be found in [23].

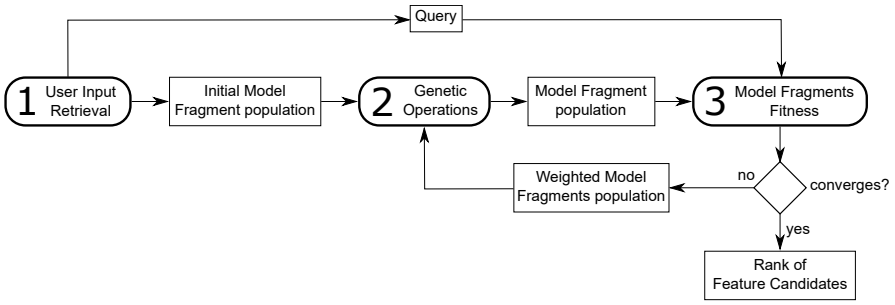

Fig. 1.2 Overview of the FLiM Approach at Design time

Fig. 1.2 shows an overview of the approach. Rounded boxes represent the different steps of the approach while rectangular boxes represent the inputs and outputs of each of the steps. Lines indicate that an element is an input or output of one of the steps.

The input of the approach is the product model where the feature is going to be located. Then, the user provides a description of the target feature in terms of an initial seed fragment and a textual description of the feature. The initial seed and the product model are used to generate some candidate fragments. Then, those candidates are assessed taking into account the textual description of the feature being located. These two steps (generation and assessment) are repeated until some stop condition is met. When the stop condition is fulfilled the process returns the list of fragment candidates ranked according to the assessments.

1.4.1 User Input

The first step is to gather input for the feature location. The input will consist of the product model and information about the target feature provided by the user. In particular, the user will provide:

- A seed fragment of the target feature is an element or set of elements that the engineer believes that could be part of the feature being located. To do so, the engineer applies her/his knowledge of the domain and the product models to point to some elements that will be used as the starting point of the process.
- A feature description of the target feature, using natural language. Typically these descriptions can come from textual documentation of the products, comments in the code, bug reports or oral descriptions from the engineers. Therefore, the query will include some domain-specific terms similar to those used when specifying the product models. The knowledge of the engineers about the domain and the product models will be useful to select the textual description from the sources available. Fig. 1.3 presents an example of input for the approach. Left part presents the seed fragment proposed by the user (a model fragment of the product model where the feature is going to be located). The user believes that the selected inductor is going to be part of the feature realization. Then, the right part of the figure shows a textual description for the feature being located, the hotplate. It is a simplified version of a text description that has been extracted from the internal documentation used by our industrial partner to describe their products.

The textual description provided by the user is turned into a query by using some well-established Information Retrieval (IR) techniques:

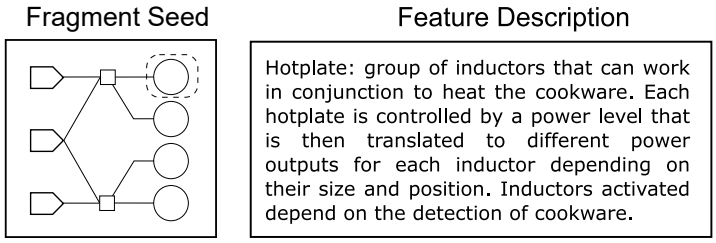

Fig. 1.3 Input provided to the approach

1. The textual description is tokenized (divided into words). Usually a white space tokenizer can be applied (that splits the strings whenever it finds a white space), but for some sources 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 can be following CamelCase naming.
2. We apply the Parts-of-Speech (POS) tagging technique. POS tagging analyses the words grammatically and infers the role of each word into the text provided. As a result, each word is tagged enabling 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 commonly used and do not contribute with relevant information that describe the feature, so they are removed.
3. Stemming techniques are applied to unify the language used in the text. This technique consists in reducing each word to its roots enabling that different words referring to similar concepts can 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 User Input Retrieval step generates as a result an initial population of fragments (that contain the model fragment provided as seed) and the query that will be used for the comparisons (obtained from the textual description). Then, the model fragment from the initial population will be evolved into several model fragments through the use of the genetic operations.

1.4.2 Encoding

We use a binary string encoding where each position of the string has two possible values: 0 or 1. However, encoding each model fragment as a string of binary values is not straightforward. Each individual of our proposed approach will be a model fragment that is defined in one of the product models. In other words, each individual is a set of model elements that are present in one of the product models.

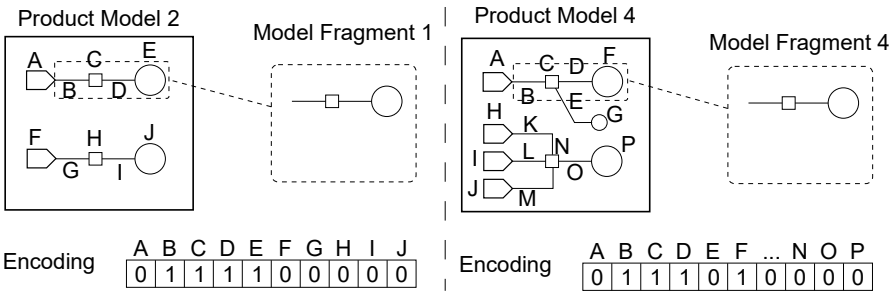

Fig. 1.4 Representation of model fragments

Fig. 1.4 shows two examples of our representation of model fragments. We denote each model element of the product model with a letter. In the example of the left part of Fig. 1.4, 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. Therefore, 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 appears in the model fragment, the value will be 1; if the model element does not appear in 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. 1.4 represent the same model fragment, but they come from different product models. Throughout the rest of the paper, we will refer to each individual as a model fragment that is part of a product model.

1.4.3 Genetic Operations

The second step is to generate a set of model fragments that could be realizing the feature. The generation of model fragments is done by applying genetic operators adapted to work over model fragments. That is, new fragments based on the existing ones (the seed fragment during the first execution) are generated through the use of three genetic operators: the selection of parents, the mutation and the crossover.

1.4.3.1 Selection of parents

In order to apply the genetic operators, it is first necessary to apply the selection operator that selects the best candidates from the population to be the input for the rest of operators. There are different methods that can be used to perform the selection of the parents, but one of the most spread

choices is to follow the wheel selection mechanism [1]. That is, each model fragment from the population has a probability of being selected proportional to their fitness score. Therefore, candidates with high fitness values will have higher probabilities of being chosen as parents for the next generation. Top part of Fig. 1.5 shows an example of application of the selection operator.

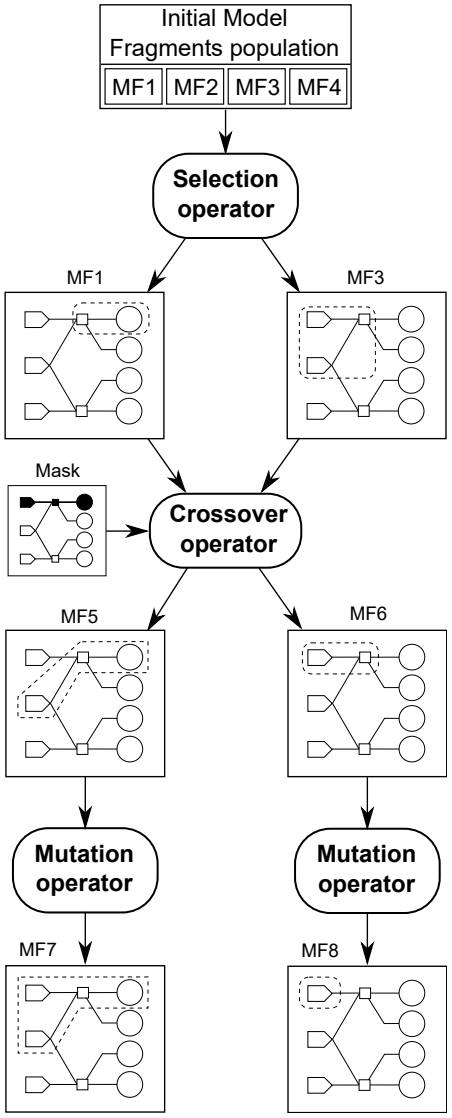

Fig. 1.5 Genetic operations: selection, crossover, and mutation over model fragments

1.4.3.2 Crossover

The crossover operator is used to imitate the sexual reproduction followed by some living beings in nature to breed new individuals. That is, two individuals mix their genomic information to give birth a new individual that holds some genetic information from one parent and some from the other one. This could make it adapt better (or worse) to her/his living environment depending on the genetic information inherited from her/his parents.

Following this idea, our crossover operator applied to model fragments takes as input two model fragments and a randomly generated mask to combine them into two new individuals. The mask determines how the combination is done, indicating for each element of the model fragments if the offspring should inherit from one parent or the other (including the element or not if the element is present on the parent or not). A model fragment is a subset of the elements present in a product model. As both model fragments have been extracted from the same product model the combination (applying the mask) of them will always return a model fragment that is part of the product model. As a result, two individuals will be generated, one by applying directly the mask and another one by applying the inverse of the mask as it is usually done in genetic algorithms [15].

Fig. 1.5 shows an example of application of the crossover operator. The input of the operator is the first parent (MF1), a mask indicating two sets of elements (one regular and one marked in black) and the second parent (MF3). To create the first of the new individuals we interpret the mask selecting the blacked out elements from the first parent (MF1) and the regular elements from the second parent (MF3). That is, the elements on the top part of the product model that are in black in this mask are selected depending on whether they are part of MF1 or not, while the rest of the elements that are not blacked out in the mask are selected depending on whether they are part of MF3 or not. As a result, the new MF5 contains some elements from the first parent (power group connected to the inductor) and some others from the second parent (the inverter that connects with the power group).

In addition, the mask is also interpreted in the opposite way, selecting the blacked out elements form the second parent and the regular elements from the first parent. This produces MF6 (see middle-right part of Fig. 1.5), where an inverter connected to a power manager has been inherited from the second parent (MF3) and nothing has been inherited from parent 1 (MF1) as all the elements not blacked out in the mask are not part of MF1.

For the crossover operation to work, it is not necessary to have elements shared by both parents. It is possible to perform crossovers that return fragments where not all the elements are connected. Indeed, the feature being located could be realized by several model elements that are not directly connected in the model. Therefore, it is necessary to create this kind of fragments as they could be the ones realizing the target feature.

1.4.3.3 Mutation

The mutation operator is used to imitate the mutations that randomly occur in nature when new individuals are born. That is, a new individual holds a small difference in regards to its parents that could make him 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 produced as output. As the approach is looking for fragments of the product model that realize a particular feature, the new modified fragment must remain being a part of the product model. Therefore, the modifications that can be done to the model fragment are driven by the product model. In particular, the mutation operator can perform two kind of modifications, addition of elements to the fragment, or removal of elements from the model fragment.

Bottom part of Fig. 1.5 shows two examples of application of the mutation operator. Left part shows the first example, MF5 is used as input of the operator that produces M7 as output. In this example, the mutation operation has added some elements (a new inverter connected to the power manager). The resulting model fragment remains being part of the product model that is driving the mutation, so it is a candidate as realization of the feature. Right part shows the second example, where MF6 is used as input and MF8 is produced as output. In this example the mutation operator has removed an element (the power manager).

1.4.4 Model Fragment Fitness

The third step of the process consists in the assessment of each candidate fragment produced and the ranking of them according to a fitness function. The fitness function is used to imitate the different degrees of adaptation to the environment that different individuals have. Therefore, individuals that result of mutations and crossovers that contribute to their adaptation to the environment will have higher chances of survival than others.

Following this idea, the fitness function is used to determine the suitability of each candidate as solution to the problem, enabling to rank them from the best candidate to the worst. The fitness function is based on the comparison between the feature description query and the identifier names and other natural language items present in the model fragments. The input of this step is a population of candidate fragments, and the feature description query; the output produced is a ranking where each candidate has been assigned with a fitness value.

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) [36] 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 [24, 40] when applied to software artifacts [28]. 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 approach, 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 in the feature description. The text from the documents (model fragments) and the text from the query (feature description) are homogenized by applying the Natural Language Processing techniques previously described in Section 1.4.1.

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.

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) [36]. 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. [35], which provides good results [61]. 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:

$$fitness(p_1) = \cos(\theta) = \frac{AB}{\|A\|\|B\|} \tag{1.1}$$

Finally, after the cosines are calculated, we obtain a value for each of the model fragments, indicating its similarity with the query.

1.4.4.1 Loop

At this point, if the stop condition is met, the process will stop returning the rank of model fragment. If the stop condition is not met yet, the genetic algorithm will keep its execution one generation more. The next time that the genetic operators are applied, it will be necessary to select the best candidates as parents for the new generation. This will be done based on the score obtained by each model fragment. As a result, model fragments with higher similarities will have more chances to be selected as parents of the new generation.

The process of generation of fragments is repeated until the stop condition is met. Usually, the stop condition can be a time slot, a fixed number of generations or a trigger value of the fitness that makes the process finish when reached. In addition, it is also possible to monitor the fitness values and determine when they are converging and no further improvements are being made by new generations. The stop condition highly depends on the domain and the problem being solved; therefore, it is adjusted depending on the results being outputted by the process.

As a result, FLiMEA provides a ranking of model fragments. The ranking is ordered following the similarity of the model fragment to the feature description, i.e. the fitness value obtained by each model fragment. Then, the domain experts will select the model fragment from the ranking, using their knowledge of the domain. At the end, the experts will be the ones working and manipulating these features as part of their daily work. Hence, they are the ones that understand and recognize them well. In a previous work [22], we conducted a usability test to know why the domain experts can discard a model fragment suggested as feature.

1.5 FLiM at Runtime (FLiMRT)

Fig. 1.6 shows an overview of our feature location approach at runtime (FLiMRT). In the Dynamic Analysis phase, the software engineer executes a scenario, which uses the target feature to be located. The information from the executed scenario is stored by means of models at runtime.

Models at runtime provide a kind of formal basis for reasoning about the current system state, for reasoning about necessary adaptations, and for analyzing the consequences of possible system adaptation. This is possible because there is a causal connection between the system and the runtime model. Then, we can use the runtime architecture model obtained from the running scenario. This model contains the elements of the model that are related to the target feature.

In the Information Retrieval phase, the approach filters the runtime architecture model to extract the relevant elements of the target feature to be

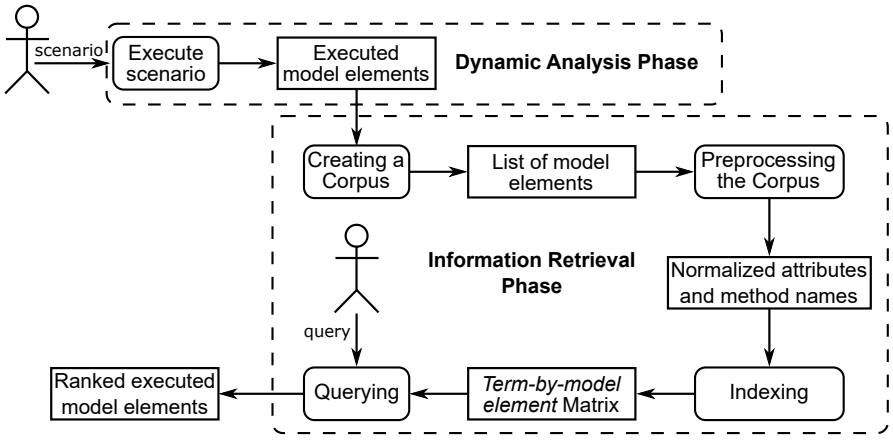

Fig. 1.6 Overview of the FLiM Approach at Runtime

located. To achieve the filtering, we adapt the same Information retrieval (IR) technique used in the design time approach, LSA [36], which allows the software engineers to write queries that describe the feature to be located. The result is a ranked list of model elements that are related to the feature based on the similarity to the provided query.

1.5.1 The Dynamic Analysis Phase

Dynamic Analysis is used to delimit the search space. Execution information is gathered via dynamic analysis (see Fig. 1.6), which is commonly used in program comprehension and involves executing a software system under specific conditions. In our case, we design scenarios that run the target features in order to obtain model traces in which the target features are involved. In other words, executing the target feature during runtime generates a feature-specific execution trace.

Our approach implies that the software engineer input is needed and of course, results are sensitive to that input. The software engineer has to decide on a scenario that will run the desired feature.

Fig. 1.7 shows the behavior of an induction hob at runtime. The induction hob is turned on in an initial configuration with a known model. In the face of changes in the context (CCs in Fig. 1.7), reconfigurations (Rs in Fig. 1.7) are triggered in order to change the configuration of the induction hob. Then, the induction hob is in a different configuration and therefore in a different model (Model Configurations in Fig. 1.7). Some examples of scenarios for different features can include putting a pot on top, the pot reaches the set temperature, the pot is moved to other place on the induction hob, or liquid spills from the

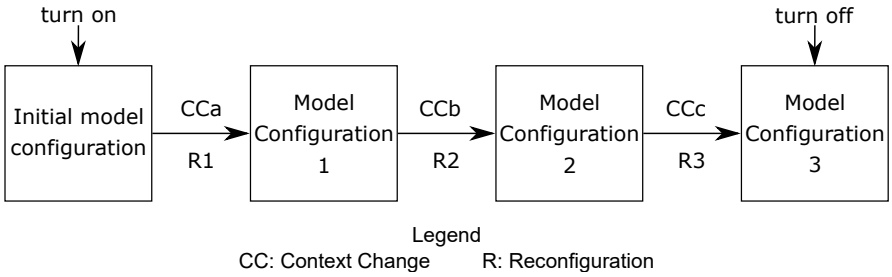

Fig. 1.7 Induction Hob at runtime

pot onto the surface. The reconfigurations activate or deactivate inductors and inverters and connect them through channels.

1.5.2 The Information Retrieval Phase

To perform Information Retrieval, we use the same technique used in the previous approach. Our approach follows the same five main steps: creating a corpus, preprocessing, indexing, querying, and generating results (see Fig. 1.6 Information Retrieval phase). We adapted each step of the LSA technique to work with architecture models. We used the architecture model that contains the executed model elements from the dynamic analysis. The adaptation is performed as follows:

- Creating a corpus. In the first step of LSA, a document granularity needs to be chosen to form a corpus. A corpus consists of a set of documents. In this approach, each document corresponds to a model element of the architecture model. Each document (model element) includes text from the names of the attributes and methods.
- Preprocessing. Once the corpus is created, it is preprocessed. Preprocessing involves normalizing the text of the documents. In this approach, the type of the attributes and the type of the parameters in the methods are removed.
- Indexing. The corpus is used to create a term-by-document matrix. Each row of the matrix corresponds to each term in the corpus, and each column represents each document. Each cell of the matrix holds a measure of the weight or relevance of the term in the document. The weight is expressed as a simple count of the number of times that the term appears in the document. In other words, each term-document pair has a number that indicates the number of times this term appears as part of the names of attributes or methods of this model element. In this work, in the term-by-document co-occurrence matrix, the terms (rows) correspond to the names of the attributes or operations (i.e., intensity) of the runtime model

and the documents (columns) correspond to the model elements that have appeared in the runtime model.

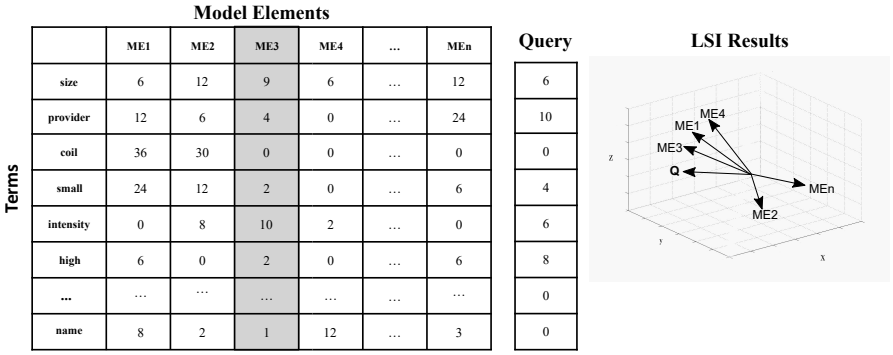

Fig. 1.8 Information Retrieval via Latent Semantic Indexing (LSI)

Fig. 1.8 shows a term-by-document co-occurrence matrix. Each row in the matrix stands for each one of the unique words (terms) extracted from the runtime model. Fig. 1.8 shows a set of representative keywords in the domain such as 'provider', 'coil', or 'intensity' as the terms of each row. Each column in the matrix stands for the model elements of the runtime model. Fig. 1.8 also shows the model elements in the columns, which represent the model elements of the runtime model. Each cell in the matrix contains the frequency with which the keyword of its row appears in the document denoted by its column. For instance, in Fig. 1.8, the term 'size' appears 9 times in the 'ME3' model element.

- Querying. A user formulates a query in natural language consisting of words or phrases that describe the feature to be located. Since LSI does not use a predefined grammar or vocabulary, users can originate queries in natural language. In this work, we use the requirements to formulate the queries. Only the relevant terms are taken into account, and words such as determinants and connectors from the language are avoided.
  In Fig. 1.8, the query column represents the words that appear in the requirement. Each cell contains the frequency with which the keyword of its row appears in the query. For instance, the term 'provider' appears 10 times in the query.
- Generating results. In LSA, the query and each document correspond to a vector. We use the same matrix factorization technique as in the design time approach. The cosine of the angle between the query vector and a document vector is used as the measure of the similarity of the document to the query. The closer the cosine is to 1, the more similar the document is to the query. A cosine similarity value is calculated between the query and each document, and then the documents are sorted by their similar-

ity values. The user inspects the ranked list to determine which of the documents are relevant to the feature.
A three-dimensional graph of the LSI results is provided in Fig. 1.8. The graph shows the representation of each one of the vectors, labeled with letters that represent the names of the model elements, which are referenced in the box below the graph. The graph reflects the 'ME3' model element vector as being the closest to the query vector, followed by the 'ME1' model element vector.

After applying the two phases of our approach, the output produced is a ranking where each model element has been assigned with a value. Only those model elements that have a similarity measure greater than x must be taken into account to measure the quality of the results. A good heuristic that is widely used is $x = 0.7$. This value corresponds to a 45% angle between the corresponding vectors. This threshold has yielded good results in other similar works [42, 58]. Determining a more generally usable heuristic for the selection of the appropriate threshold is an issue under study, over which further research is needed. The goal of our approach is to rank the relevant model elements within the top positions. The ranking of model elements is ordered by the values of the cosines.

The results obtained can be influenced by the amount of information contained in the model trace [4]. Each execution trace is related to a set of snapshots of the runtime model. However, we can have different criteria to decide when a snapshot of the runtime model should be added to the trace. For example, we could add a snapshot to the trace only when the model at run time corresponds to a valid configuration of the system, or we could add a snapshot each time a change in the architecture model at runtime is performed.

1.6 Evaluation

In this section, we present the evaluations performed to test the Feature Location in Models (FLiM) approaches. We have followed a process for evaluating each of the approaches on each of the case studies.

Fig. 1.9 shows an overview of the generic setup of the evaluation. The process is composed of: (1) an oracle obtained from our industrial partner; (2) a set of test cases extracted from the oracle; (3) an approach that is being evaluated; (4) the set of results obtained when applying the approach to the test cases; (5) the measure (or measures) that we want to evaluate; and (6) the measurements obtained based on the results yielded by the approach and the information available from the oracle.

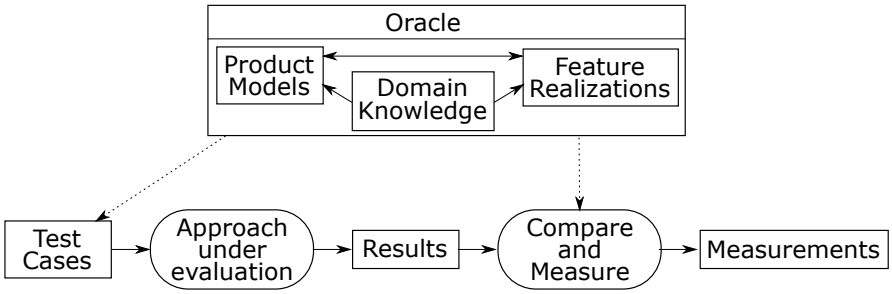

Fig. 1.9 Overview of the process followed in each evaluation

1.6.1 The Induction Hobs Domain

The first case study where we applied our approach is BSH [2]. Their induction division has been producing Induction Hobs under the brands of Bosch and Siemens for the last 15 years.

They have 46 induction hob models where, on average, each product model is composed of more than 500 elements. Their induction hobs include 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.

1.6.2 Train Control and Management Domain

The second case study where we applied our approach was CAF, a worldwide provider of railway solutions [3]. 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 receives 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 in providing the train with 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

[2] https://youtu.be/nS2sybEv6j0

[3] https://youtu.be/Ypcl2evEQB8

non-functional aspects related to regulation, such as the quality of signals from the equipment or the different levels of installed redundancy. This results in a DSL that is composed of around 1000 different elements.

As an example, the high voltage connection sequence can be described using the DSL. This high voltage 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 receive 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.

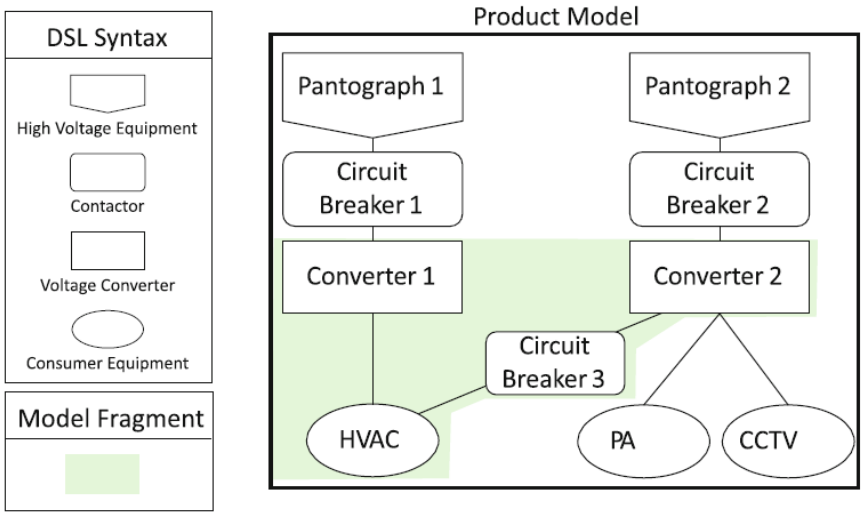

Fig. 1.10 Example of product model and model fragment (colored part) of CAF

Fig. 1.10 shows an example of a product model from a real-world train. It shows two separate pantographs (High Voltage Equipment) that 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 (the trains air conditioning system), and the PA (public address system) and CCTV (television system) on the right.

An example of model fragment is also shown in Fig. 1.10. The elements of the model fragment are highlighted in green, which are the realization of the feature: HVAC Assistance. This feature allows the passing of current from one converter to the HVAC that is assigned to its peer for coverage in case of overload or failure of the first converter.

They have 23 train models where, on average, each product model is composed of around 1200 elements. The product models are built using 121 dif-

ferent features that can be part of a specific product model. They provide us with documentation of their features and the model fragments that implement each feature.

1.6.3 Oracle Preparation

The oracle is the mechanism that we will use to evaluate the results provided by our approach. The oracle will be considered the ground truth and the results provided by the approach will be compared (when needed) with the oracle in terms of the measures that we want to obtain. In addition the oracle will be used to obtain the test cases used for the evaluation.

The oracle will be mainly composed by a set of product models and a set of features located over those product models. That is, a set of features whose realizations are model fragments and a set of product models built using those model fragments. Therefore, we have the traceability information between the features, the model fragments realizing those features, and the features being used by each product model. In addition, the oracle includes domain knowledge in different forms, such as descriptions and technical documentations for each product model, descriptions about the features, etc.

The oracle is extracted directly from the family of models of our industrial partner, that is being used to manage the products that are under production. Therefore, we consider it to be the best version available. The domain experts of each of the companies provide us with documentation and the model fragments that implements each of the features.

1.6.4 Test Cases

A set of test cases is extracted from the oracle, so the approach under evaluation can be applied to them. Fig. 1.11 shows an example of a test case. It includes a feature description in the format required by the approach (in this case, a seed fragment and a textual description of the feature) and the target product model (where the feature will be located). In addition, the test case has been extracted from the oracle and there is a corresponding model fragment for that feature description (that will be used to compare with the output provided by the approach).

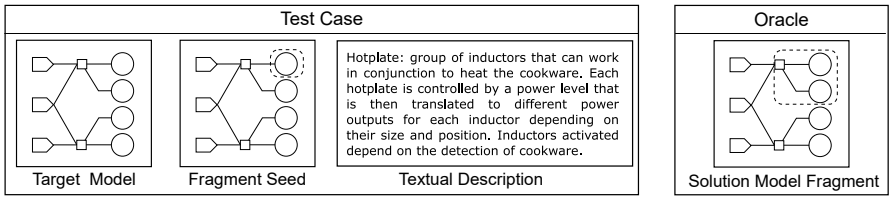

Fig. 1.11 Test case example

1.6.5 Comparison and Measure

Once the results from applying the approach to the test cases are obtained, we proceed to compare them with the oracle and measure them in terms of some software quality properties [41]. To compare the model fragments obtained and the solution from the oracle we are going to use an error matrix [62], also known as 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:

$$Precision = \frac{TP}{TP+FP} \tag{1.2}$$

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

$$Recall = \frac{TP}{TP+FN} \tag{1.3}$$

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

$$F-measure = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{1.4}$$

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:

$$MCC = \frac{TP*TN - FP*FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}} \tag{1.5}$$

1.6.6 Evaluation of FLiM at Design time (FLiMEA)

In Table 1.1, we outline the results of our algorithm in BSH and CAF. We also show the F-measure and the MCC performance indicators. The algorithm provides 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.

Table 1.1 Mean values and standard deviations for Precision, Recall, F-measure, and MCC for the Search Algorithm in BSH and CAF

	Precision $\pm\sigma$	Recall $\pm\sigma$	F-measure $\pm\sigma$	MCC $\pm\sigma$
BSH	76.47 ± 13.39	72.41 ± 13.79	72.99 ± 9.35	0.67 ± 0.13
CAF	71.75 ± 12.54	67.96 ± 15.07	68.34 ± 10.24	0.62 ± 0.13

1.6.6.1 Analysis of the results

The recall and precision values suggest that the fitness function is performing well and guiding the algorithm to find feature realization candidates close to the target feature. Following the classification of the results in precision and recall presented in [27], we can state that our approach presents excellent results (range 50% - 100%) in precision in both BSH and CAF, good results

in recall (range 70% - 79%) in BSH, and acceptable results in recall (range 60% - 69%) in CAF.

Input data The presented approach relies on two pieces of information given by the engineer performing the feature location, the seed fragment and the query. These two elements will have an impact on the ranking of model fragments produced and must be chosen carefully by the engineer performing the feature location.

To test out the impact of the seed fragment in the results, we have executed the approach with different kind of seed fragments containing one element (single element belonging to the feature being located or single element not belonging to the feature being located). But those executions did not produce noticeable differences in the resulting ranking of model fragments or in the number of generations needed to converge.

However, when selecting seed fragments of sizes closer to the size of the feature being located, the effect is noticeable. The number of generations needed by the GA to converge was reduced when a seed fragment close to the feature being located was chosen. In particular, when the fragment seed contained about 50% of the elements belonging to the feature being located, the number of generations needed for the GA to converge was reduced up to 15%.

To test out the impact of the query in the results, we have also executed the approach varying the text description used as input (using longer and smaller queries by subsetting the original description, including more or less domain terms and including more or less meta-element terms).

The search query used to locate the feature is in charge of driving the search and greatly impacts on the precision and recall results. In fact, depending on the level of detail of the query, the recall and precision values obtained will change. When the query provided is too broad, the precision decreases as there are several model elements matching the query not belonging to the target feature. Anyhow, the elements belonging to the feature will be also matched positively so the recall value will be high. However, when the query provided is too specific, some of the elements relevant for the feature being located can be missed out. Thus, the recall value is decreased although the precision values remain high.

Two examples of queries can be: (1) “A double hot plate is a group of two inductors that are heated together” and (2) “A double hotplate is formed by an inductor of small size and an inductor of big size that are connected to the same power group”. The second query is completer and more concise than the first query. Hence, the second query will obtain better results than the first one.

To achieve good precision and recall values, it is important to avoid the usage of words included into the meta-elements of the model elements. That is, if we refer to the metaclass name of one of the model elements, all instances of this class will match to that word (e.g. any inductor class model element will match the query “inductor”). By contrast, by using

words specific for the model element (as the value of the name property or values of some of the parameters contained in those model elements), those model elements (and not others with the same class) will be included into the model fragments, affecting positively to the precision values (e.g. only some inductors will match the query "doubleTwistedCoil" as it is the value of a property of the inductor class). In fact, when removing the usage of metaelement names in the queries, the approach obtained similar values of recall but the precision raised up to a 20% for best cases.

Generalization The presented approach has been designed to be applied, not only to our industrial partners domain, but to any domain. The only requisite to apply the approach is that the set of models where features have to be located conform to MOF (the OMG metalanguage for defining modeling languages). The query must be provided as a textual description.
The generation and management of fragments is performed using the Common Variability Language (CVL), which can be applied to any MOF-based languages. With the use of CVL, the approach is able to work with the model fragments provided as seed and evolve them applying the genetic algorithms. As output, the approach produces a ranking in the form of CVL model fragments.
Furthermore, the fitness function can also be applied to any MOF-based model. The text elements associated to the models are extracted automatically by the approach using the reflective methods provided by the Eclipse Modeling Framework. That is, there is no need of knowledge about the domain of application in order to extract the relevant terms.
However, the approach can be tailored to fit the needs of different domains if necessary. For instance, the naming conventions used by companies for model elements, properties and functions can follow different formats, but the approach can be tailored to handle them. In our case studies some model elements follow the CamelCase convention while others follow the Underscore convention. To address that, we applied different tokenizers in order to obtain the terms properly. Similarly, the Part-of-Speech tagger that is used to eliminate non-relevant words based on their grammatical category is language dependant, but can be configured to other languages when necessary.
In summary, the approach can be applied to locate features on any MOF-based model from any domain. If necessary, some tweaks and modifications can be applied to tailor the approach to particular needs of the domains, but the core of the approach will remain unchanged.

1.6.7 Evaluation of FLiM at Runtime (FLiMRT)

In this case, instead of using precision, recall, F-measure and MCC, we evaluate the position of the first model element in the ranking. It is accepted by

the feature location community [39, 56] that a feature location approach is considered better than another feature location when it produces a ranking where the elements that belongs to the feature are in higher positions than in the ranking of the other approach.

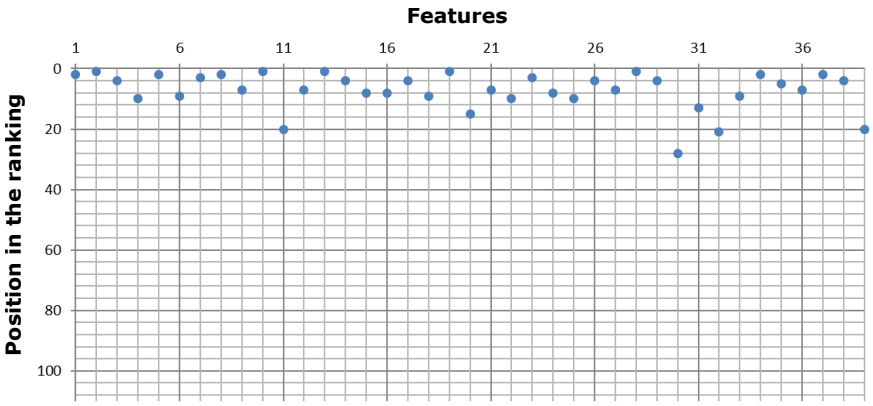

Fig. 1.12 Position of the first model element that belong to the target feature in the ranking for each one of the features

We performed this evaluation with thirty-nine features of BSH. We defined the scenarios based on bug reports of each one of the features. Fig. 1.12 shows the position of the first model element that belong to the target feature in the ranking for each one of the thirty-nine features. The x-axis represents the features, and the y-axis represents the position in the ranking. The blue dots represent the first model element for each feature. The position of the first model element that belongs to each one of the features has values between 1 and 28, where the 84% of the results are in the top ten positions.

1.6.7.1 Analysis of the results

Fig. 1.13 shows the graphical representation of the ranking for one feature (feature number five in Fig. 1.12). Due to space constraints, we only show the graphical representation for one feature, however, all the rankings follow a similar distribution in the results.

The query is the vector that is on the x-axis. The remainder of the vectors are model elements. Those that have been tagged by the oracle have a r_i label at the end of the arrow, while those that have not been tagged have nothing at the end of the arrow. The angle corresponds to the cosine with which we calculated the position in the ranking (see Section 1.5.2); the closer the model element is to the query, the higher the position in the ranking. The length of each vector indicates the number of times that the terms appear in each

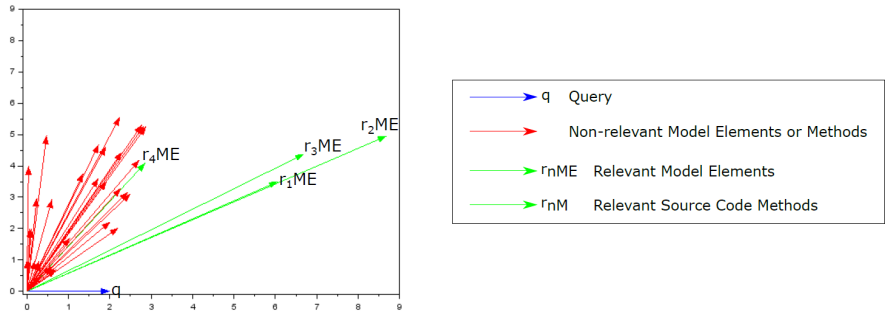

Fig. 1.13 Vectorial representation of the model elements in the Ranking of one feature

model element. The longer the vector is, the more terms appear in the model element. The graph only shows the thirty-three which have positive cosines, the rest, thirteen, are in the left of the *y*-axis and have few relevance for the query.

The goal of a feature location technique is to reduce the effort required by software engineers to find the desired feature. Our approach on average requires searching in less than fifty model elements while a source-code-based approach on average requires searching in more than three thousand eight hundred methods.

The graphical representation of Fig. 1.13 allows us to see that the approach performs a good discrimination between relevant and non-relevant model elements. The majority of the model elements that belong to the feature achieve better results than the ones that do not belong.

Since architecture models at runtime allow working on a high level of abstraction, the words used at the model level are closer to the query. By raising the level of abstraction with the architecture model, we can prevent auxiliary methods and variables from interfering with the feature location.

Finally, we realized that the model elements that contained few attributes and methods got worse positions in the ranking than the ones that contained more attributes and methods. For example, one of the elements related to the feature in Fig. 1.13 obtained position 27 in the ranking. This is because this element corresponds to a channel element. This particular channel only has three attributes that describe the information that goes through the channel. The information required by this element was not as detailed as the other model elements when specifying the model. For this reason, the model element corresponding to this channel got a lower position in the ranking. In contrast, other kinds of channels got better positions since, on average, they have about twenty attributes and methods.

1.7 Conclusion

In this chapter, we presented approaches for Feature Location in Models (FLiM) at design time (FLiMEA) and at runtime (FLiMRT). Specifically, we have presented a Genetic Algorithm for FLiM at design time, and an approach that combines architecture models and information retrieval for FLiM at runtime.

Software systems are becoming more increasingly complex, and systems with models are not an exception. Hence, software maintenance is becoming more and more important. In particular, the feature location area has gained significant attention and cannot be neglected in the models area.

The results show that our design time (FLiMEA) and runtime approaches (FLiMRT) can be applied to address the challenge of feature location in models (FLiM). Specifically, the use of genetic operations for models provided good results in our studies. In addition, this demonstrates that FLiMEA and FLiMRT for feature location at the model level can be applied in real world environments.

References

1. Affenzeller, M., Winkler, S., Wagner, S., Beham, A.: Genetic Algorithms and Genetic Programming: Modern Concepts and Practical Applications, 1^{th} edn. Chapman & Hall/CRC (2009)
2. Alves, V., Schwanninger, C., Barbosa, L., Rashid, A., Sawyer, P., Rayson, P., Pohl, C., Rummler, A.: An exploratory study of information retrieval techniques in domain analysis. In: 2008 12th International Software Product Line Conference, pp. 67–76 (2008). DOI 10.1109/SPLC.2008.18
3. Arcega, L., Font, J., Haugen, Ø., Cetina, C.: Feature location through the combination of run-time architecture models and information retrieval. In: J. Grabowski, S. Herbold (eds.) System Analysis and Modeling. Technology-Specific Aspects of Models : 9th International Conference, SAM 2016, Saint-Malo, France, October 3-4, 2016. Proceedings, pp. 180–195. Springer International Publishing (2016). DOI 10.1007/978-3-319-46613-2_12
4. Arcega, L., Font, J., Haugen, Ø., Cetina, C.: On the influence of models at run-time traces in dynamic feature location. In: Modelling Foundations and Applications - 13th European Conference, ECMFA 2017, Held as Part of STAF 2017, Marburg, Germany, July 19-20, 2017, Proceedings (2017)
5. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling step-wise refinement. In: Proceedings of the 25th International Conference on Software Engineering, ICSE '03, pp. 187–197. IEEE Computer Society, Washington, DC, USA (2003). URL http://dl.acm.org/citation.cfm?id=776816.776839
6. Berger, T., Rublack, R., Nair, D., Atlee, J.M., Becker, M., Czarnecki, K., Wąsowski, A.: A survey of variability modeling in industrial practice. In: Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems, VaMoS '13, pp. 7:1–7:8. ACM, New York, NY, USA (2013). DOI 10.1145/2430502.2430513. URL http://doi.acm.org/10.1145/2430502.2430513
7. Biggerstaff, T.J., Mitbander, B.G., Webster, D.: The concept assignment problem in program understanding. In: Proceedings of the 15th International Conference on

Software Engineering, ICSE '93, pp. 482–498. IEEE Computer Society Press, Los Alamitos, CA, USA (1993). URL http://dl.acm.org/citation.cfm?id=257572.257679
8. Blei, D.M., Ng, A.Y., Jordan, M.I., Lafferty, J.: Latent dirichlet allocation. Journal of Machine Learning Research 3(4/5), 993 – 1022 (2003)
9. Chastek, G., McGregor, J.: Guidelines for developing a product line production plan. Tech. Rep. CMU/SEI-2002-TR-006, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (2002)
10. Chawla, I., Singh, S.K.: Performance evaluation of vsm and lsi models to determine bug reports similarity. In: 2013 Sixth International Conference on Contemporary Computing (IC3), pp. 375–380 (2013). DOI 10.1109/IC3.2013.6612223
11. Chen, K., Rajlich, V.: Case study of feature location using dependence graph. In: Proceedings IWPC 2000. 8th International Workshop on Program Comprehension, pp. 241–247 (2000). DOI 10.1109/WPC.2000.852498
12. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns, 3rd edn. SEI Series in Software Engineering. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2001)
13. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA (2000)
14. Czarnecki, K., Wasowski, A.: Feature diagrams and logics: There and back again. In: 11th International Software Product Line Conference (SPLC 2007), pp. 23–34 (2007). DOI 10.1109/SPLINE.2007.24
15. Davis, L.: Handbook of Genetic Algorithms. Van Nostrand Reinhold, New York (1991)
16. Dit, B., Revelle, M., Gethers, M., Poshyvanyk, D.: Feature location in source code: a taxonomy and survey. Journal of Software: Evolution and Process 25(1), 53–95 (2013). DOI 10.1002/smr.567
17. Donohoe, P.: Proceedings of the 1st International Software Product Lines Conference (SPLC 2000). ISBN 0-7923-7940-3. Denver, Colorado, USA (2000)
18. Duszynski, S., Knodel, J., Becker, M.: Analyzing the source code of multiple software variants for reuse potential. In: 2011 18th Working Conference on Reverse Engineering, pp. 303–307 (2011). DOI 10.1109/WCRE.2011.44
19. Eaddy, M., Aho, A., Antoniol, G., Gueheneuc, Y.G.: Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In: Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on, pp. 53–62 (2008). DOI 10.1109/ICPC.2008.39
20. Eisenbarth, T., Koschke, R., Simon, D.: Locating features in source code. IEEE Transactions on Software Engineering 29(3), 210–224 (2003). DOI 10.1109/TSE.2003.1183929. URL http://dx.doi.org/10.1109/TSE.2003.1183929
21. Eisenberg, A., De Volder, K.: Dynamic feature traces: finding features in unfamiliar code. In: Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on, pp. 337–346 (2005)
22. Font, J., Arcega, L., Haugen, Ø., Cetina, C.: Building software product lines from conceptualized model patterns. In: Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015, pp. 46–55 (2015). DOI 10.1145/2791060.2791085. URL https://doi.org/10.1145/2791060.2791085
23. Font, J., Arcega, L., Haugen, Ø., Cetina, C.: 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, MODELS '16, pp. 272–282. ACM, New York, NY, USA (2016). DOI 10.1145/2976767.2976789. URL http://doi.acm.org/10.1145/2976767.2976789
24. Gethers, M., Oliveto, R., Poshyvanyk, D., Lucia, A.D.: On integrating orthogonal information retrieval methods to improve traceability recovery. In: IEEE 27th International Conference on Software Maintenance, ICSM 2011, Williamsburg, VA, USA,

September 25-30, 2011, pp. 133–142 (2011). DOI 10.1109/ICSM.2011.6080780. URL https://doi.org/10.1109/ICSM.2011.6080780
25. Harman, M.: 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, FASE'10, pp. 1–12. Springer-Verlag, Berlin, Heidelberg (2010). DOI 10.1007/978-3-642-12029-9. URL http://dx.doi.org/10.1007/978-3-642-12029-9
26. Haugen, Ø., Møller-Pedersen, B., Oldevik, J., Olsen, G.K., Svendsen, A.: Adding standardized variability to domain specific languages pp. 139–148 (2008). DOI 10.1109/SPLC.2008.25. URL https://doi.org/10.1109/SPLC.2008.25
27. Hayes, J.H., Dekhtyar, A., Sundaram, S.K.: Advancing candidate link generation for requirements tracing: the study of methods. IEEE Transactions on Software Engineering 32(1), 4–19 (2006). DOI 10.1109/TSE.2006.3
28. Hindle, A., Barr, E.T., Su, Z., Gabel, M., Devanbu, P.: On the naturalness of software. In: Proceedings of the 34th International Conference on Software Engineering, ICSE '12, pp. 837–847. IEEE Press, Piscataway, NJ, USA (2012). URL http://dl.acm.org/citation.cfm?id=2337223.2337322
29. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.: Feature-oriented domain analysis (foda) feasibility study. Tech. Rep. CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (1990). URL http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=11231
30. Kästner, C., Dreiling, A., Ostermann, K.: Variability mining: Consistent semi-automatic detection of product-line features. IEEE Transactions on Software Engineering 40(1), 67–82 (2014). DOI 10.1109/TSE.2013.45
31. Kästner, C., Giarrusso, P.G., Rendel, T., Erdweg, S., Ostermann, K., Berger, T.: Variability-aware parsing in the presence of lexical macros and conditional compilation. In: Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '11, pp. 805–824. ACM, New York, NY, USA (2011). DOI 10.1145/2048066.2048128. URL http://doi.acm.org/10.1145/2048066.2048128
32. Kästner, C., Ostermann, K., Erdweg, S.: A variability-aware module system. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '12, pp. 773–792. ACM, New York, NY, USA (2012). DOI 10.1145/2384616.2384673. URL http://doi.acm.org/10.1145/2384616.2384673
33. Koschke, R., Quante, J.: On dynamic feature location. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, ASE '05, pp. 86–95. ACM, New York, NY, USA (2005). DOI 10.1145/1101908.1101923. URL http://doi.acm.org/10.1145/1101908.1101923
34. Krueger, C.W.: Easing the transition to software mass customization. In: Revised Papers from the 4th International Workshop on Software Product-Family Engineering, PFE '01, pp. 282–293. Springer-Verlag, London, UK, UK (2002). URL http://dl.acm.org/citation.cfm?id=648114.748909
35. Kuhn, A., Ducasse, S., Gírba, T.: Semantic clustering: Identifying topics in source code. Inf. Softw. Technol. 49(3), 230–243 (2007). DOI 10.1016/j.infsof.2006.10.017. URL http://dx.doi.org/10.1016/j.infsof.2006.10.017
36. Landauer, T.K., Foltz, P.W., Laham, D.: An introduction to latent semantic analysis. Discourse Processes 25(2-3), 259–284 (1998). DOI 10.1080/01638539809545028
37. van der Linden, F. (ed.): Software Product-Family Engineering, 4th International Workshop, PFE 2001, Bilbao, Spain, October 3-5, 2001, Revised Papers, Lecture Notes in Computer Science, vol. 2290. Springer (2002). DOI 10.1007/3-540-47833-7. URL https://doi.org/10.1007/3-540-47833-7
38. Liu, D., Marcus, A., Poshyvanyk, D., Rajlich, V.: 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, ASE '07, pp. 234–243. ACM, New York, NY, USA (2007). DOI 10.1145/1321631.1321667. URL http://doi.acm.org/10.1145/1321631.1321667

39. Liu, D., Marcus, A., Poshyvanyk, D., Rajlich, V.: 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, ASE '07, pp. 234–243. ACM, New York, NY, USA (2007). DOI 10.1145/1321631.1321667. URL http://doi.acm.org/10.1145/1321631.1321667
40. Lucia, A., Penta, M., Oliveto, R., Panichella, A., Panichella, S.: Labeling source code with information retrieval methods: An empirical study. Empirical Softw. Engg. 19(5), 1383–1420 (2014). DOI 10.1007/s10664-013-9285-5. URL http://dx.doi.org/10.1007/s10664-013-9285-5
41. Manning, C.D., Raghavan, P., Schütze, H.: Introduction to Information Retrieval. Cambridge University Press, New York, NY, USA (2008)
42. Marcus, A., Sergeyev, A., Rajlich, V., Maletic, J.: An information retrieval approach to concept location in source code. In: Proceedings of the 11th Working Conference on Reverse Engineering, pp. 214–223 (2004). DOI 10.1109/WCRE.2004.10
43. Martinez, J., Ziadi, T., Bissyandé, T.F., Klein, J., l. Traon, Y.: 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, pp. 396–406 (2015). DOI 10.1109/ASE.2015.44
44. Martinez, J., Ziadi, T., Bissyandé, T.F., Le Traon, Y.: Bottom-up adoption of software product lines: A generic and extensible approach. In: Proceedings of the 19th International Software Product Line Conference, SPLC '15. Nashville, TN, USA. (2015)
45. McIlroy, M.D.: Mass-produced software components. In: J.M. Buxton, P. Naur, B. Randell (eds.) Software Engineering Concepts and Techniques (1968 NATO Conference of Software Engineering), pp. 88–98. NATO Science Committee (1968)
46. Nadi, S., Berger, T., Kästner, C., Czarnecki, K.: Mining configuration constraints: Static analyses and empirical results. In: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, pp. 140–151. ACM, New York, NY, USA (2014). DOI 10.1145/2568225.2568283. URL http://doi.acm.org/10.1145/2568225.2568283
47. Niu, N., Easterbrook, S.: On-demand cluster analysis for product line functional requirements. In: 2008 12th International Software Product Line Conference, pp. 87–96 (2008). DOI 10.1109/SPLC.2008.11
48. Northrop, L.M.: Sei's software product line tenets. IEEE Softw. 19(4), 32–40 (2002). DOI 10.1109/MS.2002.1020285. URL http://dx.doi.org/10.1109/MS.2002.1020285
49. OMG: Common variability language (CVL), OMG revised submission 2012. OMG document: ad/2012-08-05 (2012)
50. Parnas, D.L.: On the design and development of program families. IEEE Trans. Softw. Eng. 2(1), 1–9 (1976). DOI 10.1109/TSE.1976.233797. URL http://dx.doi.org/10.1109/TSE.1976.233797
51. Pérez, F., Font, J., Arcega, L., Cetina, C.: Collaborative feature location in models through automatic query expansion. Autom. Softw. Eng. 26(1), 161–202 (2019). DOI 10.1007/s10515-019-00251-9. URL https://doi.org/10.1007/s10515-019-00251-9
52. Pohl, K., Böckle, G., Linden, F.J.v.d.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2005)
53. Poshyvanyk, D., Gueheneuc, Y.G., Marcus, A., Antoniol, G., Rajlich, V.: Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. IEEE Transactions on Software Engineering 33(6), 420–432 (2007). DOI 10.1109/TSE.2007.1016. URL http://dx.doi.org/10.1109/TSE.2007.1016

54. Rahman, M.M., Chakraborty, S., Ray, B.: Which similarity metric to use for software documents?: A study on information retrieval based software engineering tasks. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE '18, pp. 335–336. ACM, New York, NY, USA (2018). DOI 10.1145/3183440.3194997. URL http://doi.acm.org/10.1145/3183440.3194997
55. Revelle, M., Dit, B., Poshyvanyk, D.: Using data fusion and web mining to support feature location in software. In: IEEE 18th International Conference on Program Comprehension (ICPC), pp. 14–23 (2010). DOI 10.1109/ICPC.2010.10
56. Revelle, M., Dit, B., Poshyvanyk, D.: Using data fusion and web mining to support feature location in software. In: Program Comprehension (ICPC), 2010 IEEE 18th International Conference on, pp. 14–23 (2010). DOI 10.1109/ICPC.2010.10
57. Rubin, J., Chechik, M.: A survey of feature location techniques. In: I. Reinhartz-Berger, A. Sturm, T. Clark, S. Cohen, J. Bettin (eds.) Domain Engineering, pp. 29–58. Springer Berlin Heidelberg (2013). DOI 10.1007/978-3-642-36654-3_2
58. Salman, H.E., Seriai, A., Dony, C.: Feature location in a collection of product variants: Combining information retrieval and hierarchical clustering. In: The 26th International Conference on Software Engineering and Knowledge Engineering, Hyatt Regency, Vancouver, BC, Canada, July 1-3, 2013., pp. 426–430 (2014)
59. Salton, G., McGill, M.J.: Introduction to Modern Information Retrieval. McGraw-Hill, Inc., New York, NY, USA (1986)
60. She, S., Lotufo, R., Berger, T., Wasowski, A., Czarnecki, K.: Reverse engineering feature models. In: Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, pp. 461–470. ACM, New York, NY, USA (2011). DOI 10.1145/1985793.1985856. URL http://doi.acm.org/10.1145/1985793.1985856
61. van der Spek, P., Klusener, S., van de Laar, P.: Complementing Software Documentation, pp. 37–51. Springer Netherlands, Dordrecht (2011). DOI 10.1007/978-90-481-9849-8_3
62. Stehman, S.V.: Selecting and interpreting measures of thematic classification accuracy. Remote Sensing of Environment 62(1), 77–89 (1997). DOI 10.1016/S0034-4257(97)00083-7. URL http://www.sciencedirect.com/science/article/pii/S0034425797000837
63. Svendsen, A., Zhang, X., Lind-Tviberg, R., Fleurey, F., Haugen, Ø., Møller-Pedersen, B., Olsen, G.K.: Developing a software product line for train control: A case study of cvl. In: Proceedings of the 14th International Conference on Software Product Lines: Going Beyond, SPLC'10, pp. 106–120. Springer-Verlag, Berlin, Heidelberg (2010). URL http://dl.acm.org/citation.cfm?id=1885639.1885650
64. Thomas, S.W., Hassan, A.E., Blostein, D.: Mining Unstructured Software Repositories, pp. 139–162. Springer Berlin Heidelberg, Berlin, Heidelberg (2014). DOI 10.1007/978-3-642-45398-4
65. Wilde, N., Scully, M.C.: Software reconnaissance: Mapping program features to code. Journal of Software Maintenance 7(1), 49–62 (1995)
66. Wong, W.E., Gokhale, S.S., Horgan, J.R., Trivedi, K.S.: Locating program features using execution slices. In: Proceedings 1999 IEEE Symposium on Application-Specific Systems and Software Engineering and Technology. ASSET'99 (Cat. No.PR00122), pp. 194–203 (1999). DOI 10.1109/ASSET.1999.756769
67. Zhang, X., Haugen, Ø., Møller-Pedersen, B.: Model comparison to synthesize a model-driven software product line. In: 15th International Software Product Line Conference (SPLC), pp. 90–99 (2011). DOI 10.1109/SPLC.2011.24