UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Definition of Descriptive and Diagnostic Measurements for Model Fragment Retrieval

July 2021

Author: Manuel Ballarin Naya

Directors: Dr. Vicente Pelechano Farragut
Dr. Carlos Cetina Englada

Dedication

To my family. Thank you mom for being a father and a mother throughout my life. Thank you for giving me security against danger, thank you for giving me self-confidence against hardships, and thank you for enlightening me when I needed it most. To my sisters. Thank you Lucia for teaching me that no matter how long the storm lasts, the sun always shines behind the clouds. Thank you Cristina for teaching me that life gives me a 24-hour check every day, I decide how to invest it.

To Mari. You are incredible. I have been looking for you all my life. Thank you for being there when I least deserve it, because that is when I really need it.

To my partners. Raúl, Ana, Lorena, Jaime, Paqui, Carlos, Jorge. Without your endless endeavour, this book would have never been achieved. You have been the architects of this thesis, without you nothing of it would have been possible. Thank you friends.

Bear in mind that the wonderful things you learn in your schools are the work of many generations, produced by enthusiastic effort and infinite labour in every country of the world.
The World As I see It, Albert Einstein.

Acknowledgments

First of all, I want to thank my directors, Dr. Carlos Cetina and Dr. Vicente Pelechano, without whom this thesis would have never been possible to achieve. I know that I have been a difficult and a different student for you. Thank you for having risen to the challenge. There is no amount of words that can even begin to express my gratitude for your patience and guidance.

I also want to express my gratitude to my coworkers (now, friends) in the SVIT Research Group and in USJ. This was not an easy road for me at all times, and they have always been there for me. Thank you for your support in the hardest moments, and for the all the shared good memories as well.

In addition, I cannot finish this section without thanking my family for all the unconditional help towards achieving my goals, and for their valuable life teachings and wise advice. I could not have walked this road without them.

Abstract

Throughout the pages of this document, I present the results of the research that was carried out in the context of my PhD studies.

Nowadays, software exists in almost everything. Companies often develop and maintain a collection of custom-tailored software systems that share some common features but also support customer-specific ones. As the number of features and the number of product variants grows, software maintenance is becoming more and more complex. To keep pace with this situation, Model-Based Software Engineering Community is addressing a key-activity: Model Fragment Location (MFL). MFL aims at identifying model elements that are relevant to a requirement, feature, or bug. Many MFL approaches have been introduced in the last few years to address the identification of the model elements that correspond to a specific functionality. However, there is a lack of detail when the measurements about the search space (models) and the measurements about the solution to be found (model fragment) are reported. The goal of this thesis is to provide insights to MFL Research Community of how to improve the report of location problems. We propose using five measurements (size, volume, density, multiplicity, and dispersion) to report the location problems during MFL. The usage of these novel measurements support researchers during the creation of new MFL approaches and during the improvement of those existing ones. Using two different case

studies, both real and industrial, we emphasize the importance of these measurements in order to compare results in a deeply way. The results of the research have been redacted and published in forums, conferences, and journals specialized in the topics and context of the research.

This thesis is presented as compendium of articles according the regulations in Universitat Politècnica de València. This thesis document introduces the topics, context, and objectives of the research, presents the academic publications that have been published as a result of the work, and then discusses the outcomes of the investigation.

Resumen

A través de las páginas de este documento, presento los resultados de la investigación realizada en el contexto de mis estudios de doctorado.

Hoy en día, el software existe en casi todo. Las empresas a menudo desarrollan y mantienen colecciones de sistemas de software personalizados que comparten algunas características entre ellos, pero que también tienen otras características particulares. Conforme el número de características y el número de variantes de un producto crece, el mantenimiento del software se vuelve cada vez más complejo. Para hacer frente a esta situación la Comunidad de Ingeniería del Software basada en Modelos está abordando una actividad clave: la Localización de Fragmentos de Modelo. Esta actividad consiste en la identificación de elementos del modelo que son relevantes para un requisito, una característica o un bug.

Durante los últimos años se han propuesto muchos enfoques para abordar la identificación de los elementos del modelo que corresponden a una funcionalidad en particular. Sin embargo, existe una carencia a la hora de cómo se reportan las medidas del espacio de búsqueda, así como las medidas de la solución a encontrar. El objetivo de nuestra tesis radica en proporcionar a la comunidad dedicada a la actividad de localización de fragmentos de modelo una serie de medidas (tamaño, volumen, densidad, multiplicidad y dispersión) para reportar los problemas de localización de fragmentos de modelo.

El uso de estas novedosas medidas ayuda a los investigadores durante la creación de nuevos enfoques, así como la mejora de aquellos enfoques ya existentes. Mediante el uso de dos casos de estudio reales e industriales, esta tesis pone en valor la importancia de estas medidas para comparar resultados de diferentes enfoques de una manera precisa. Los resultados de este trabajo han sido redactados y publicados en foros, conferencias y revistas especializadas en los temas y contexto de la investigación.

Esta tesis se presenta como un compendio de artículos acorde a la regulación de la Universitat Politècnica de València. Este documento de tesis presenta los temas, el contexto y los objetivos de la investigación. Presenta las publicaciones académicas que se han publicado como resultado del trabajo y luego analiza los resultados de la investigación.

Resum

A través de les pàgines d'aquest document, presente els resultats de la investigació realitzada en el context dels meus estudis de doctorat.

Hui en dia, el programari existix en quasi tot. Les empreses sovint desenrotllen i mantenen col·leccions de sistemes de programari personalitzats que compartixen algunes característiques entre ells, però que també tenen altres característiques particulars. Conforme el nombre de característiques i el nombre de variants d'un producte creix, el manteniment del programari es torna cada vegada més complex. Per a fer front a esta situació la Comunitat d'Enginyeria del Programari basada en Models està abordant una activitat clau: la Localització de Fragments de Model. Esta activitat consistix en la identificació d'elements del model que són rellevants per a un requisit, una característica o un bug.

Durant els últims anys s'han proposat molts enfocaments per a abordar la identificació dels elements del model que corresponen a una funcionalitat en particular. No obstant això, hi ha una carència a l'hora de com es reporten les mesures de l'espai de busca, així com les mesures de la solució a trobar. L'objectiu de la nostra tesi radica a proporcionar a la comunitat dedicada a l'activitat de localització de fragments de model una sèrie de mesures (grandària, volum, densitat, multiplicitat i dispersió) per a reportar els problemes de localització de fragments de model.

L'ús d'estes noves mesures ajuda als investigadors durant la creació de nous enfocaments, així com la millora d'aquells enfocaments ja existents. Per mitjà de l'ús de dos casos d'estudi reals i industrials, esta tesi posa en valor la importància d'estes mesures per a comparar resultats de diferents enfocaments d'una manera precisa. Els resultats d'este treball han sigut redactats i publicats en fòrums, conferències i revistes especialitzades en els temes i context de la investigació.

Esta tesi es presenta com un compendi d'articles d'acord amb la regulació de la Universitat Politècnica de València. Este document de tesi presenta els temes, el context i els objectius de la investigació. Presenta les publicacions acadèmiques que s'han publicat com resultat del treball i després analitza els resultats de la investigació.

Contents

Part I

Introduction

Introduction

This chapter introduces this thesis, highlighting the motivation for the research and the objectives of this work, as well as providing an overview of the thesis and of the scientific articles included in this compendium document. Finally, this chapter presents the methodology followed to pursue this research, and states the structure of the document.

Motivation

Nowadays, software exists in almost everything. Companies often develop and maintain a collection of custom-tailored software systems that share some common features but also support different, customer-specific ones. As the number of features and the number of product variants grows, software maintenance becomes more and more complex. Software maintenance is an emerging discipline that reduces engineering costs and Time-to-Market as well as increases the quality of individual developments. Lehman et al. (Lehman, Ramil, and Kahen 2001) pointed out that up to 80% of the lifetime of a system is spent on maintenance and evolution activities. Software maintainers spend from 50% up to almost 90% of their time trying to understand a program in order to make changes correctly.

To keep pace with this situation, there are three key activities that the Model-Based Software Engineering Community is addressing: Feature Location (FL), Traceability Links Recovery (TLR), and Bug Location (BL). FL is known as the process of finding the set of software artifacts that realize a specific functionality (Dit et al. 2013). TLR is concerned with the ability to relate artefacts created during the development of a software system to describe the system from different perspectives and levels of abstraction with each other (Spanoudakis and Zisman 2005). BL aims to identify the location of the artifact that is pertinent to a software fault (Arcega, Jaime Font, Haugen, and Cetina - 2019). Addressing these activities is critical to the success of a project, leads to increased maintainability and reliability of software systems, and decreases the expected defect rate in developed software. When these activities (FL, TLR, and BL) are conducted on a model-based family of software products, they are encompassed in the term *Model Fragment Location (MFL).*

Model Fragment Location (MFL) aims at identifying model elements that are relevant to a requirement, feature, or bug. From the timeless traceability activity (Winkler and Pilgrim 2010), (P. Mäder and A. Egyed 2012), (Jaber, Sharif, and Liu 2013), (Patrick Mäder and Alexander Egyed 2014) to recent research efforts on Feature Location (David Binkley and Dawn Lawrie 2010), (Rubin and Chechik 2013), (Dit et al.

2013), (Martinez et al. 2015), (Jaime Font, Arcega, et al. 2016), (J. Font et al. 2017) and Bug Location (Zhang et al. 2016), (Kusumoto et al. 2002), (Arcega, Jaime Font, Haugen, and Cetina 2017), MFL has been gaining momentum.

Tens of MFL works can be found today. For instance, Winkler et al. (Winkler and Pilgrim 2010) classify several approaches that have been created in the past 15 years which try to optimize the automatic identification of traces in models. De Lucia et al. (De Lucia et al. 2004) present a TLR method and tool, which are based on Latent Semantic Indexing (LSI) and include models. Spanoudakis et al. (Spanoudakis, Zisman, et al. 2004) present a linguistic rule-based approach to support the automatic generation of Traceability Links between requirements and models. More recently, Lapeña el al. (Lapeña Martí et al. 2017) presented CACAO4M, an approach for ranking relevant model fragments for the development of specific requirements for a new product. Font et al. (Jaime Font, Arcega, et al. 2016) presented a Genetic Algorithm to Feature Location. In (Arcega, Jaime Font, Haugen, and Cetina 2017) the authors proposed an approach for bug localization in models (BLiM2). Specifically, there is an emerging interest in leveraging machine learning techniques to address the challenges of MFL (D. Binkley and D. Lawrie 2014), (Corley, Damevski, and Kraft 2015),(B. Le et al. 2016), (Ana C Marcén, Pérez, and Cetina 2017), (Ana C. Marcén et al. 2017),(Mills, Escobar-Avila, and Haiduc 2018).

Prior works leverage Information Retrieval, Linguistic techniques, and Search-based techniques to achieve the location of relevant model fragments. The aim of these works is focused on providing new MFL approaches or to propose improvements of those existing ones. Nonetheless, all of these works have one thing in common: none of them place their interest on how to report their results deeply. There is a lack of detail when the measurements about the search space (models) and the measurements about the solution space (model fragment) are reported. Generally, the only reported measure is the model size. However, in most of the cases, the model size values are not comparable among different works since different models are measured in different ways.

One major problem for integrating study results into a common body of knowledge is the heterogeneity of reporting styles. Firstly, it is difficult to locate relevant information because the same type of information is located in different sections of different study reports and secondly, important information is often missing - for example, context information is reported differently and without taking into account further generalizability (Jedlitschka, Ciolkowski, and Pfahl 2008). The same problem applies to current MFL works. In most cases, the results of different MFL works are not comparable among them since different models are measured in different ways, being difficult to locate relevant information because the results are reported differently. It is not the same challenge to locate a large model fragment in a small model than to locate a small and scattered model fragment over several large models. Properly reporting the location problem is an important challenge that has not get enough attention yet.

Thesis Objectives

The goal of this thesis is to provide insights to the MFL research community on how to improve the report of location problems. The work carried out to date has focused on proposing new MFL approaches or on improving those existing ones by providing the algorithms and the parameters to tune them in detail. In contrast, our work is focused on providing a set of measurements that are strongly related with models to improve the location problems, enabling researchers to create new MFL approaches and to improve their existing ones. More specifically, our work contributes to the MFL research community by providing real measurements of location problems from the real world, which can be used as a reference for the community when creating synthetic location problems. The fact of knowing the influence of our measurements on the results in advance enables us to limit the information that must be taken into account in experiment planning, also empowering us to report more accurate results during these experiments.

The research community has identified different types of measurements depending on the intended use; these are descriptive, diagnostic, predictive, and prescriptive measurements (Delen and Ram 2018). Descriptive measurements describe what happened in the past. Diagnostic measurements help with understanding why something happened in the past. Predictive measurements predict what is most likely to happen in the future. Prescriptive measurements recommend actions you can take to affect those outcomes. Regarding our proposal, size and volume are descriptive measurements which measure the search space (models), while density, multiplicity, and dispersion are diagnostic measurements which measure the solution space (model fragments). Properly reporting the location problem is important because otherwise it is not possible to compare the results of different works with each other.

Of our proposed measurements, **size** measures the number of elements that the model contains. Since the larger the model, the larger the search space, this measurement determines how complex the search space ends up being in order to find the solution. **Volume** measures the number of models that compose the search space where a solution is searched for. Since the larger the number of models, the larger the search space, this measurement determines how large the search space becomes based on the number of models. **Density** measures the percentage of model elements that realize a solution. In other words, since the model fragment is composed of the model elements that realize the solution, the density is computed as the ratio of model fragment elements to model elements. Since the larger the model fragment, the larger the density, this measurement determines how large the solution ends up being with respect to the model. **Multiplicity** measures the number of times the solution appears in the search space. Since the more solutions found, the greater the multiplicity, this measurement determines how complex the search ends up being, based on the number of solutions that the search space contains for the same solution. Finally, **dispersion** measures the ratio of connected elements in the solution. Dispersion is computed as the ratio between the number of groups and the number of elements. Since the more groups found, the larger the dispersion, this measurement determines how complex the search ends up being depending on whether or not the model elements that compose a model fragment are linked.

In order to determine the relevance of the proposed measurements, we studied whether the values of the measurements have an impact on the results by pushing it to industrial settings. We evaluated our proposal in two different case studies, both real and industrial. The first one is based on an Information Retrieval (IR) approach for Feature Location (FL) (see chapter 5), and the second one is based on a Machine Learning-based (ML) approach for FL (see chapter 6). Both provide promising results on how to deeply report the location problems during MFL. Figure 1 shows a quick reference about the scope of the work done as part of this thesis. It has been divided in order to establish clearly which elements constitute the background, which are part of the thesis work, and which are infrastructure for that work.

Based on the context provided in the motivation section, the main goal of this thesis is to discuss why model size measurement (the widespread measurement to report MFL) does not provide enough information on the location problem, and why our proposed measurements are significant for the research community. To that extent, the following research questions are defined for the thesis:

RQ1 Determine whether, and to what extent, our proposed measurements influence Information Retrieval (IR) Model Fragment Location approaches.

RQ2 Determine whether, and to what extent, our proposed measurements influence Machine Learning-based (ML) Model Fragment Location approaches.

The results of the research carried out to fulfill the pursued objectives have been published in several articles, introduced in the following section.

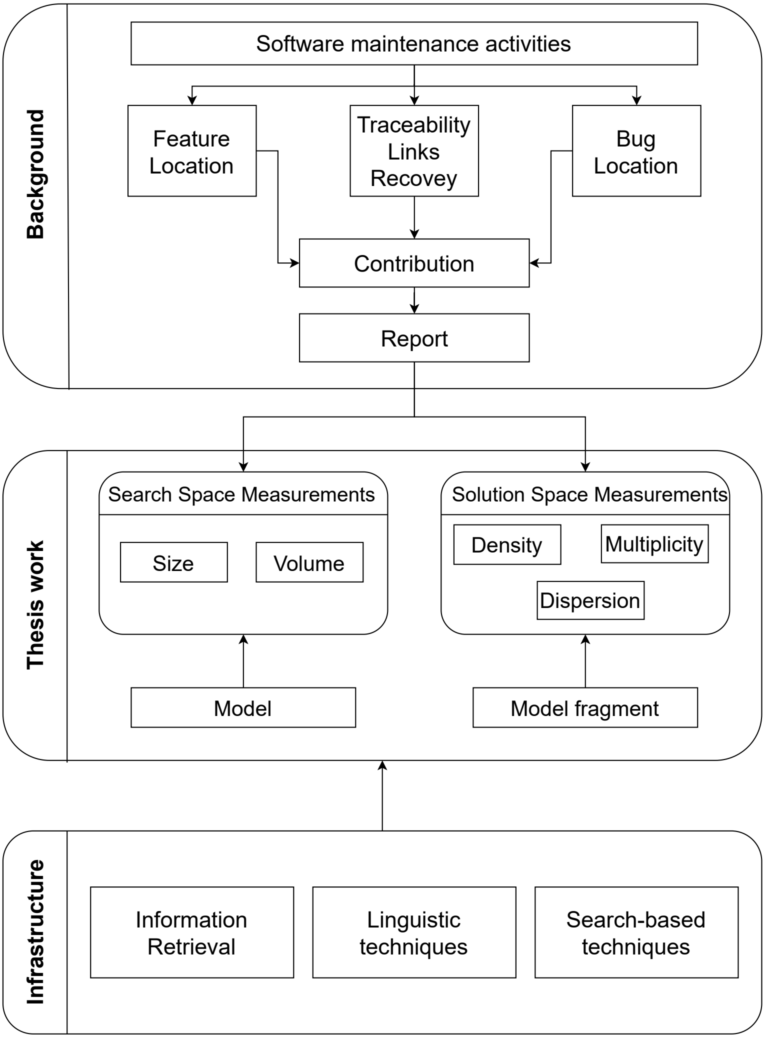


Figure 1: Cheat Sheet

Thesis Overview

Figure 2 presents an overview of the works that we carried out in order to respond to the research questions posed by the thesis. We started the thesis by identifying the most novel, state-of-the-art approaches in feature location, and concentrated our initial efforts in transforming the approaches towards their application to model-based software artifacts. We were able to automate the variability formalization of a given family of models (provided by one of our industrial partners) into a Software Product Line, publishing the results of this work in a paper for REVE SPLC '15 (Jaime Font, Ballarin, et al. 2015). Later, focusing on the requirements level, we published a paper for SPLC '16 (Lapeña, Ballarin, and Cetina 2016). This paper leverages Part-of-Speech tagging and Latent Semantic Indexing to rank the relevancy of legacy products for a new development at the requirements level, and to locate their most significant methods for each of the new product requirements.

After that, we successfully leveraged feature location and code-comparison techniques identifying the clone-and-own relationships between the same features in different model-based product variants. We published the results of the work derived from our initial ideas in a paper for ICSR '16 (Ballarin, Lapeña Martí, and Cetina 2016). In addition, this work was extended and published for IEEE Access '18 (Pérez et al. 2018) presenting significant differences with our previous work, such as the application of the approach in a different industrial domain and a further extension of the evaluation to measure the performance of our approach in terms of recall and precision values.

During the development of our first research ideas, we realized the importance of deeply reporting feature location results. Our first works present novelty approaches which assist software engineers to build a SPL from their model-based software products, advise them to the best way of locating the most relevant methods to each requirement of the new product, and help them by suggesting improvements on feature reuse. Regardless of their novelty, none of them reported their results deeply.

Having said that, we began to realize that model size was not enough in order to report the location problem during Model Fragment Location

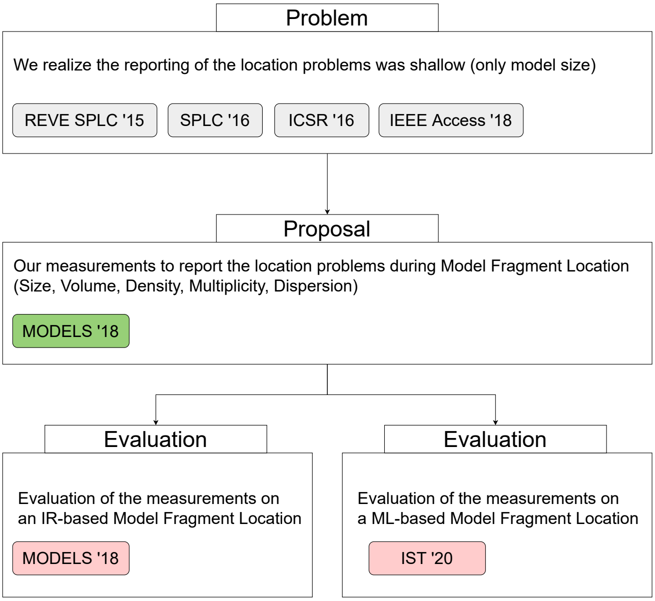

Problem
We realize the reporting of the location problems was shallow (only model size)
REVE SPLC '15
SPLC '16
ICSR '16
IEEE Access '18
Proposal
Our measurements to report the location problems during Model Fragment Location (Size, Volume, Density, Multiplicity, Dispersion)
MODELS '18
Evaluation
Evaluation of the measurements on an IR-based Model Fragment Location
MODELS '18
Evaluation
Evaluation of the measurements on a ML-based Model Fragment Location
IST '20

Figure 2: Thesis work overview

(MFL). It did not matter how original a particular approach was, nor by whom this approach was created, if it was not reporting its results in a proper way. In general terms, members of the research community focused in MFL were using the model size measure as the only way to report their experimental results. Hence, we identified the need for taking into account other measurements to report the location problem during MFL as a clearly emergent research issue. Therefore, we focused our efforts on exploring the particularities of reporting the location problems during Model Fragment Location.

The MFL research community cannot consider their reporting measures as a sufficient posture. They must consider other measurements to report the location problems during MFL. Having said that, we were in charge of providing the usage of measurements to report the location problems during MFL. We started exploring the particularities of how

MFL approaches report the results of their experiments. In addition, we studied the different types of measurements proposed by the research community with the aim of understanding them and analyzing which measurements are best suited to report software experiments. At this time, our emerging interest on how to report location problems started to gather pace.

We propose using five measurements to report the results of MFL techniques. Of the five proposed measurements, size and volume (both descriptive measurements) measure the search space, while density, multiplicity, and dispersion (all of them diagnostic measurements) measure the solution. We can confirm that our proposal shows that all the proposed measurements have a direct impact on the results of MFL approaches. In order to confirm our expectations, we conducted an evaluation with our industrial partner CAF on a real-world Information Retrieval (IR) approach. The results of this work, which was published in MODELS '18 (Ballarin, Marcén, et al. 2018), brought to light certain particularities of the research challenge related on how to deeply report the location problems.

Later, during the final stage of this thesis, we have worked towards assessing the impact of the proposed measurements on a real Machine Learning-based approach for feature location. Our results show that model size is not the only measurement that has an impact on the feature location results. Model Fragment Location approaches should analyze the influence of our measurements on their case studies not only to properly report their results but also to be able to compare the approaches fairly, improving the feature location results of their case studies.

Thesis Structure

This thesis is conformed and presented as compendium of articles. According to the guidelines and regulations for the development of a PhD thesis in Universitat Politècnica de València, a PhD thesis that is presented as a compendium of articles must be structured in four parts:

I Introduction: The first part of the thesis (Part I) introduces the motivation for the research, the description of the problem along with the objectives of the work, the list of scientific articles published towards the fulfillment of the thesis goals, and the methodology that was followed to pursue the research presented in this thesis.

II Publications: The second part of the thesis (Part II, chapters 1 to 6) provides the compendium of scientific articles that result from the research that was carried out for the thesis. The contributions are ordered chronologically and adapted to the format of the thesis.

III Results: The third part of the thesis (Part III) discusses the results and contributions of the thesis to the research context, and the future works that arise as a continuation of the ongoing research.

IV Conclusions: The fourth and final part of the thesis (Part IV) finishes the thesis by providing a few concluding remarks for the presented work.

The following section presents more information on the compendium of articles included in this thesis, and on their relationship with the research questions, research projects, and case studies.

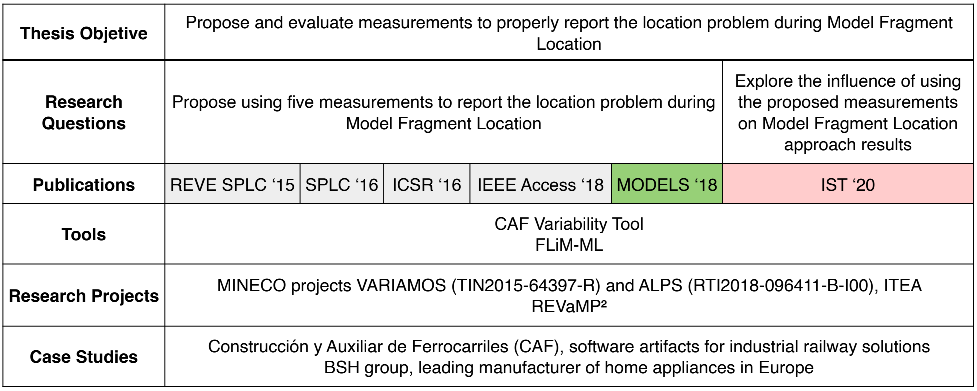
Articles Compendium

Figure 3 comprises a general overview of the research works that have been carried out as a result of this thesis. In the figure, it is possible to appreciate a total of six rows:

- The first row states the main objective of the thesis.
- The second row introduces the Research Questions posed by this thesis.
- The third row links the publications with the research questions posed by the thesis.
- The fourth row presents the frameworks and tools which have been developed as a result of our research.

- The fifth row presents the research projects that served as a framework for the research.

- Finally, the fifth and final row indicates the industrial and academic case studies to which the thesis research has been applied.



Thesis Objetive	Propose and evaluate measurements to properly report the location problem during Model Fragment Location					
Research Questions	Propose using five measurements to report the location problem during Model Fragment Location					Explore the influence of using the proposed measurements on Model Fragment Location approach results
Publications	REVE SPLC '15	SPLC '16	ICSR '16	IEEE Access '18	MODELS '18	IST '20
Tools	CAF Variability Tool FLiM-ML					
Research Projects	MINECO projects VARIAMOS (TIN2015-64397-R) and ALPS (RTI2018-096411-B-I00), ITEA REVaMP²					
Case Studies	Construcción y Auxiliar de Ferrocarriles (CAF), software artifacts for industrial railway solutions BSH group, leading manufacturer of home appliances in Europe					

Figure 3: Thesis contributions

Figure 4 shows a roadmap of this thesis. It consists of six chapters corresponding to the research articles that have been developed as a result of the research that has been carried out for this thesis. We provide three different ways to read the chapters of this thesis. The first one is a full version which includes all the research articles that have been developed (represented in the figure with a solid line). The second one is to read a brief version of our work (see the dotted line in the figure). The third one, which allows the reader to grasp the results of the thesis, is to read the CORE version (represented with the dashed line in the figure).

1. **Automating the variability formalization of a model family by means of common variability language (REVE SPLC '15)** (Jaime Font, Ballarin, et al. 2015): In this paper we present Model Family to SPL, an approach to automate the variability formalization of a given family of models into a SPL. In particular, our approach enables the automatic decomposition of new product models into model fragments that are incorporated to the model library.

2. **Towards clone-and-own support: locating relevant methods in legacy products (SPLC '16)** (Lapeña, Ballarin, and Cetina 2016): In this paper, we propose a novel approach, named Computer Assisted CAO (CACAO), that leverages Part-of-Speech tagging and adapts Latent Semantic Indexing to rank the relevancy of legacy products for a new development at the requirements level, and to locate their most significant methods for each of the new product requirements.

3. **Leveraging Feature Location to Extract the Clone-and-Own Relationships of a Family of Software Products (ICSR '16)** (Ballarin, Lapeña Martí, and Cetina 2016): In this paper, we presented an approach to locate clone-and-own relationships between features in model-based families of software products. We evaluated our approach in the industrial domain of Induction Hobs (IH) over two families of IH products. On one of them, the firmware code of the products was implemented manually from the models. On the other, the firmware code of the products was implemented in an automatic way.

4. **Locating Clone-and-Own Relationships in Model-based Industrial Families of Software Products to Encourage Reuse (IEEE ACCESS '18)** (Pérez et al. 2018): Through this paper, we extended our prior work through the modification of the step of our approach that compares the source code of a feature in a product with the source code of the same feature in another product in order to avoid irrelevant textual differences; the application of our approach in a different industrial domain (the train control PLC software provided by CAF) in order to prove its generalization; and an extension of the evaluation to measure the performance of both our approach and the baseline (the previous version of our approach) in terms of recall and precision in all the industrial case studies in use.

5. **Measures to report the Location Problem of Model Fragment Location (MODELS '18)** (Ballarin, Marcén, et al. 2018): In this paper, we propose using five measurements (size, volume, density, multiplicity, and dispersion) to report the location prob-

lem during Model Fragment Location. In order to determine the relevance of the proposed measurements, we studied whether the values of the measurements have an impact on the results provided by two distinct MFL approaches. Our results show that all of the proposed measurements have a direct impact on the results of MFL approaches.

6. **On the influence of Model Fragment Properties in Machine Learning-based Feature Location (IST '20)** (Ballarín et al. 2021): In prior work, we have proposed using five measurements to report the location problems. Through this paper, we explore the influence of three of the measurements (density, multiplicity, and dispersion) on a machine learning-based approach for feature location. The analysis of the results shows that both the density and the dispersion measurements significantly influence the results.

The works that conform this thesis have formed part of the research context of three funding projects: two Spanish national research plans named VARIAMOS (TIN2015-64397-R) and ALPS (RTI2018-096411-8-I00), devoted to the extraction of software variability in Software Product Lines, and one international project from the second call of a European ITEA 3 project named REVaMP^2, devoted to the creation of a holistic platform and process for variability extraction. The approaches proposed as a result of this research have been validated through two case studies: (1) a proprietary industrial case study provided by one of our industrial partners, CAF (Construcciones y Auxiliar de Ferrocarriles), manufacturer of railway solutions, and (2) an industrial case study provided by another of our industrial partners, BSH Group, leading manufacturer of home appliances in Europe. The various works presented in this thesis have led to the development of a series of frameworks and tools. One of those tools, the Train Control and Management Variability Tool, can be seen in action at: youtube.com/watch?v=Ypcl2evEQB8

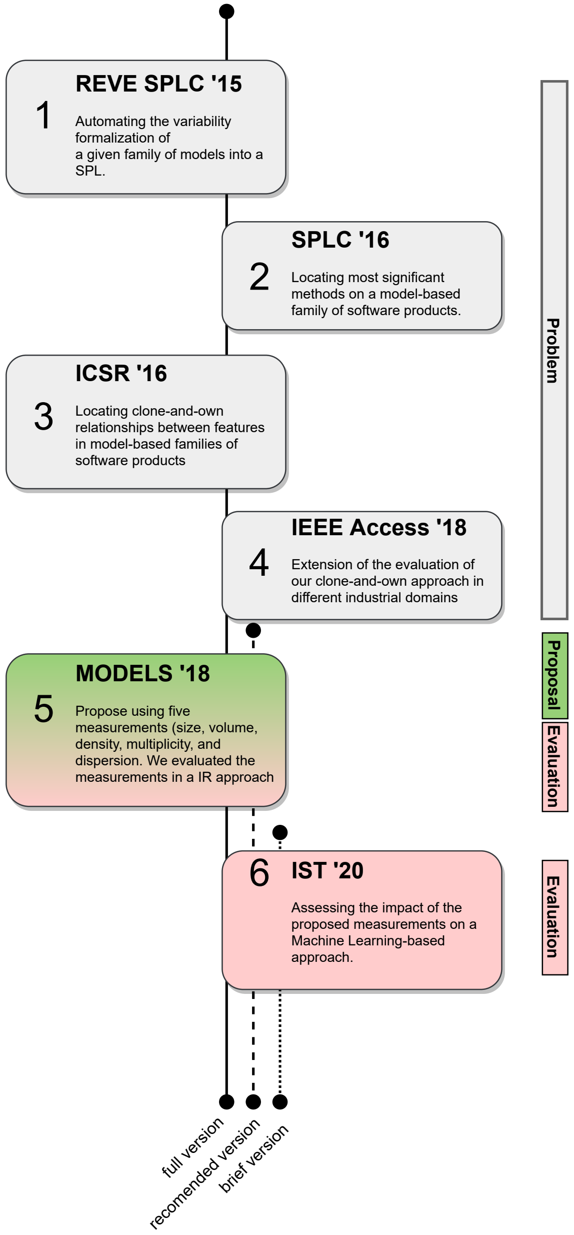

REVE SPLC '15
1
Automating the variability formalization of
a given family of models into a SPL.
SPLC '16
2
Locating most significant methods on a model-based family of software products.
ICSR '16
3
Locating clone-and-own relationships between features in model-based families of software products
IEEE Access '18
4
Extension of the evaluation of our clone-and-own approach in different industrial domains
MODELS '18
5
Propose using five measurements (size, volume, density, multiplicity, and dispersion. We evaluated the measurements in a IR approach
6
IST '20
Assessing the impact of the proposed measurements on a Machine Learning-based approach.
Problem
Proposal
Evaluation
Evaluation
full version
recomended version
brief version

Figure 4: Thesis Roadmap

Research Methodology

For the development of this thesis, we have followed the Action-research Methodology (Baskerville 1999). Action-research is a collaborative type of research which seeks to make theory and practice meet, to establish a link between research and practice by means of a cyclical process. Action-Research focuses on yielding new knowledge which is useful in practice. It is gained by introducing changes and by researching into candidate solutions to different real scenarios which are relevant to a group in practice (Avison et al. 1999). This is achieved thanks to the intervention of a researcher in the real circumstances surrounding the group. The results of these experiences must be beneficial to both the researcher and the participants. In recent years, there is an increasing tendency towards the use of Action-research in Software Engineering to address different research topics (Santos and Travassos 2009).

Action-research does not refer to a specific research method, but rather to a set of methods of the same type which share the following properties: (i) Focus on action and change; (ii) Focus on a problem, (iii) An "organic" process model which involves systematic and interactive phases, and (iv) Participants' collaboration. An outline of the use of Action-research in Information Systems is provided in (Lau 1997), including several examples published by different authors regarding the analysis, design and development of Information Systems, and particularly on software implementation and related processes.

Following the guidelines proposed in the methodology, we identify our research as a research cycle breaking down the generic activities in order to deeply report the location problem during Model Fragment Location by providing and evaluating different measurements, not only to properly report but also to be able to compare the approaches fairly and thus improve the feature location results. We applied the methodology as follows 5:

1. **Diagnosing**: in this phase, we brought to light the main research problems, and formulated the appropriate research questions.

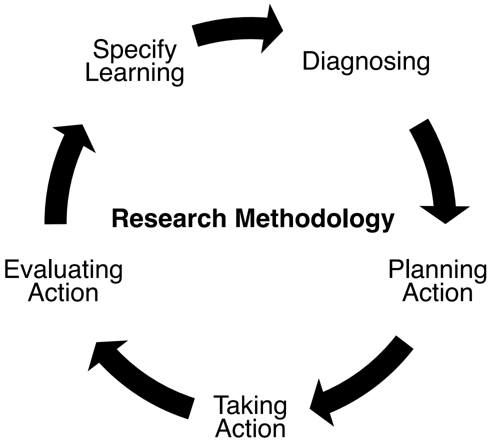


Figure 5: Research cycle

2. **Planning Action**: in this phase, we studied the validity of the approaches through incorporating measurements commonly accepted by the research community and extensively used in the relevant literature. In addition, we studied the threats to the validity of the approaches and how to mitigate them.

3. **Taking Action**: in this phase, we developed the approaches and applied them to the case studies to obtain results, which have been embodied into several research articles.

4. **Evaluating Action**: in this phase, we analyzed the obtained results, obtaining responses to the posed research questions and identifying novel research opportunities in the process.

5. **Specify Learning**: in this phase, we analyzed the obtained results, obtaining responses to the posed research questions and identifying novel research opportunities in the process.

The cyclic process described in Figure 5 has been applied in an iterative fashion. In this thesis, the first cycle started by transporting Feature Location approaches from code-based software artifacts towards model-based software artifacts. The responses to the research questions posed by the initial challenge triggered novel research questions, which acted as starting points for further research.

Bibliography

Arcega, Lorena, Jaime Font, Oystein Haugen, and Carlos Cetina (2017). "On the Influence of Models at Run-Time Traces in Dynamic Feature Location". In: *Modelling Foundations and Applications.* Ed. by Anthony Anjorin and Huáscar Espinoza. Cham: Springer International Publishing, pp. 90–105. ISBN: 978-3-319-61482-3. DOI: 10.1007/978-3-319-61482-3_6 (cit. on p. 5).

Arcega, Lorena, Jaime Font, Oystein Haugen, and Carlos Cetina - (2019). "An approach for bug localization in models using two levels: model and metamodel". In: *Software and Systems Modeling* 18.6, pp. 3551–3576. ISSN: 1619-1374. DOI: 10.1007/s10270-019-00727-y (cit. on p. 4).

Avison, David E. et al. (Jan. 1999). "Action Research". In: *Commun. ACM* 42.1, pp. 94–97. ISSN: 0001-0782. DOI: 10.1145/291469.291479 (cit. on p. 18).

B. Le, Tien-Duy et al. (2016). "A Learning-to-rank Based Fault Localization Approach Using Likely Invariants". In: *Proceedings of the 25th International Symposium on Software Testing and Analysis.* ISSTA 2016. Sarrebruck, Germany: ACM, pp. 177–188. ISBN: 978-1-4503-4390-9. DOI: 10.1145/2931037.2931049 (cit. on p. 5).

Ballarin, Manuel, Raúl Lapeña Martí, and Carlos Cetina (June 2016). "Leveraging Feature Location to Extract the Clone-and-Own Relationships of a Family of Software Products". In: pp. 215–230. ISBN: 978-3-319-35121-6. DOI: 10.1007/978-3-319-35122-3_15 (cit. on pp. 10, 15).

Ballarin, Manuel, Ana Marcén, et al. (Oct. 2018). "Measures to report the Location Problem of Model Fragment Location". In: *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering*

Languages and Systems, pp. 189–199. DOI: `10.1145/3239372.3239397` (cit. on pp. 12, 15).

Ballarín, Manuel et al. (2021). “On the influence of model fragment properties on a machine learning-based approach for feature location”. In: *Information and Software Technology* 129, p. 106430. ISSN: 0950-5849. DOI: `https://doi.org/10.1016/j.infsof.2020.106430` (cit. on p. 16).

Baskerville, Richard L (1999). “Investigating information systems with action research”. In: *Communications of the association for information systems* 2.1, p. 19 (cit. on p. 18).

Binkley, D. and D. Lawrie (Sept. 2014). “Learning to Rank Improves IR in SE”. In: *2014 IEEE International Conference on Software Maintenance and Evolution*. Washington, DC, USA: IEEE, pp. 441–445. DOI: `10.1109/ICSME.2014.70` (cit. on p. 5).

Binkley, David and Dawn Lawrie (2010). “Maintenance and Evolution: Information Retrieval Applications”. In: DOI: `10.1081/E-ESE-120044704` (cit. on p. 4).

Corley, C. S., K. Damevski, and N. A. Kraft (Sept. 2015). “Exploring the Use of Deep Learning for Feature Location”. In: *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Washington, DC, USA: IEEE, pp. 556–560. DOI: `10.1109/ICSM.2015.7332513` (cit. on p. 5).

De Lucia, A. et al. (2004). “Enhancing an artefact management system with traceability recovery features”. In: *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* Pp. 306–315. DOI: `10.1109/ICSM.2004.1357816` (cit. on p. 5).

Delen, Dursun and Sudha Ram (2018). “Research challenges and opportunities in business analytics”. In: *Journal of Business Analytics* 1.1, pp. 2–12. DOI: `10.1080/2573234X.2018.1507324`. eprint: `https://doi.org/10.1080/2573234X.2018.1507324` (cit. on p. 7).

Dit, Bogdan et al. (Jan. 2013). “Feature location in source code: A taxonomy and survey”. In: *Journal of Software Maintenance and Evolution: Research and Practice* 25. DOI: 10.1002/smr.567 (cit. on p. 4).

Font, J. et al. (2017). “Achieving Feature Location in Families of Models through the use of Search-Based Software Engineering”. In: *IEEE Transactions on Evolutionary Computation*, pp. 1–1. ISSN: 1089-778X. DOI: 10.1109/TEVC.2017.2751100 (cit. on p. 5).

Font, Jaime, Lorena Arcega, et al. (2016). “Feature Location in Model-Based Software Product Lines Through a Genetic Algorithm”. In: *International Conference on Software Reuse*. Springer, pp. 39–54. ISBN: 978-3-319-35122-3. DOI: 10.1007/978-3-319-35122-3_3 (cit. on p. 5).

Font, Jaime, Manuel Ballarin, et al. (July 2015). “Automating the variability formalization of a model family by means of common variability language”. In: pp. 411–418. DOI: 10.1145/2791060.2793678 (cit. on pp. 10, 14).

Jaber, K., B. Sharif, and C. Liu (2013). “A Study on the Effect of Traceability Links in Software Maintenance”. In: *IEEE Access* 1, pp. 726–741. DOI: 10.1109/ACCESS.2013.2286822 (cit. on p. 4).

Jedlitschka, Andreas, Marcus Ciolkowski, and Dietmar Pfahl (2008). “Reporting Experiments in Software Engineering”. In: *Guide to Advanced Empirical Software Engineering*. Ed. by Forrest Shull, Janice Singer, and Dag I. K. Sjøberg. London: Springer London, pp. 201–228. ISBN: 978-1-84800-044-5. DOI: 10.1007/978-1-84800-044-5_8 (cit. on p. 6).

Kusumoto, Shinji et al. (Mar. 2002). “Experimental Evaluation of Program Slicing for Fault Localization”. In: *Empirical Software Engineering* 7, pp. 49–76. DOI: 10.1023/A:1014823126938 (cit. on p. 5).

Lapeña, Raúl, Manuel Ballarin, and Carlos Cetina (2016). “Towards Clone-and-Own Support: Locating Relevant Methods in Legacy Products”. In: *Proceedings of the 20th International Systems and Software Product Line Conference*. SPLC ’16. Beijing, China: Association for Computing Machinery, pp. 194–203. ISBN: 9781450340502. DOI: 10.1145/2934466.2934485 (cit. on pp. 10, 15).

Lapeña Martí, Raúl et al. (June 2017). "Model Fragment Reuse Driven by Requirements". In: (cit. on p. 5).

Lau, F. (1997). "A Review on the Use of Action Research in Information Systems Studies". In: *Information Systems and Qualitative Research: Proceedings of the IFIP TC8 WG 8.2 International Conference on Information Systems and Qualitative Research, 31st May–3rd June 1997, Philadelphia, Pennsylvania, USA*. Ed. by Allen S. Lee, Jonathan Liebenau, and Janice I. DeGross. Boston, MA: Springer US, pp. 31–68. ISBN: 978-0-387-35309-8. DOI: 10.1007/978-0-387-35309-8_4 (cit. on p. 18).

Lehman, M. M., J. F. Ramil, and G. Kahen (2001). *A Paradigm for the Behavioural Modelling of Software Processes using System Dynamics*. Tech. rep. (cit. on p. 4).

Mäder, P. and A. Egyed (2012). "Assessing the effect of requirements traceability for software maintenance". In: *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pp. 171–180. DOI: 10.1109/ICSM.2012.6405269 (cit. on p. 4).

Mäder, Patrick and Alexander Egyed (Apr. 2014). "Do developers benefit from requirements traceability when evolving and maintaining a software system?" In: *Empir Software Eng* 20, pp. 1–29. DOI: 10.1007/s10664-014-9314-z (cit. on p. 4).

Marcén, Ana C, Francisca Pérez, and Carlos Cetina (2017). "Ontological Evolutionary Encoding to Bridge Machine Learning and Conceptual Models: Approach and Industrial Evaluation". In: *International Conference on Conceptual Modeling*. Cham, Switzerland: Springer, pp. 491–505. DOI: 10.1007/978-3-319-69904-2_37 (cit. on p. 5).

Marcén, Ana C. et al. (2017). "Towards Feature Location in Models Through a Learning to Rank Approach". In: *Proceedings of the 21st International*

Systems and Software Product Line Conference - Volume B. SPLC '17. Sevilla, Spain: ACM, pp. 57–64. ISBN: 978-1-4503-5119-5. DOI: `10.1145/3109729.3109734` (cit. on p. 5).

Martinez, J. et al. (Nov. 2015). "Automating the Extraction of Model-Based Software Product Lines from Model Variants (T)". In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 396–406. DOI: `10.1109/ASE.2015.44` (cit. on p. 5).

Mills, C., J. Escobar-Avila, and S. Haiduc (Sept. 2018). "Automatic Traceability Maintenance via Machine Learning Classification". In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Washington, DC, USA: IEEE, pp. 369–380. DOI: `10.1109/ICSME.2018.00045` (cit. on p. 5).

Pérez, F. et al. (2018). "Locating Clone-and-Own Relationships in Model-Based Industrial Families of Software Products to Encourage Reuse". In: *IEEE Access* 6, pp. 56815–56827. DOI: `10.1109/ACCESS.2018.2873509` (cit. on pp. 10, 15).

Rubin, Julia and Marsha Chechik (May 2013). "A Survey of Feature Location Techniques". In: DOI: `10.1007/978-3-642-36654-3_2` (cit. on p. 4).

Santos, Paulo Sergio Medeiros dos and Guilherme Horta Travassos (2009). "Action research use in software engineering: An initial survey". In: *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pp. 414–417. DOI: `10.1109/ESEM.2009.5316013` (cit. on p. 18).

Spanoudakis, George and Andrea Zisman (Aug. 2005). "Software Traceability: A Roadmap". In: *Handbook of Software Engineering and Knowledge Engineering* 3. DOI: `10.1142/9789812775245_0014` (cit. on p. 4).

Spanoudakis, George, Andrea Zisman, et al. (2004). "Rule-based generation of requirements traceability relations". In: *Journal of Systems and Software* 72.2, pp. 105–127. ISSN: 0164-1212. DOI: `https://doi.org/10.1016/S0164-1212(03)00242-5` (cit. on p. 5).

Winkler, Stefan and Jens Pilgrim (Sept. 2010). “A survey of traceability in requirements engineering and model development”. In: *Software and System Modeling* 9, pp. 529–565. DOI: `10.1007/s10270-009-0145-0` (cit. on pp. 4, 5).

Zhang, T. et al. (2016). “A Literature Review of Research in Bug Resolution: Tasks, Challenges and Future Directions”. In: *The Computer Journal* 59.5, pp. 741–773. DOI: `10.1093/comjnl/bxv114` (cit. on p. 5).

Part II

Compendium of Scientific Articles

Chapter 1

Automating the Variability Formalization of a Model Family By Means of Common Variability Language

The aim of domain engineering process is to define and realise the commonality and variability of a Software Product Line. In the context of a family of models, spotting the commonalities and differences may become cumbersome and error prone as the number of models and its complexity increases. This work presents an approach to automate the formalization of variability in a given family of models. As output, the variability is made explicit in terms of Common Variability Language. The model commonalities and differences are specified as placements over a base model and replacements in a model library. The resulting Software Product Line (SPL) enables the derivation of new product models by reusing the extracted model fragments. Furthermore, the SPL can be evolved by the creation of new models, which are in turn automatically decomposed as model fragments of the SPL. The approach has been validated with our industrial partner (BSH), an induction hobs company. Finally, we present five different evolution scenarios encountered during the validation.

1.1 Introduction

A Software Product Line (SPL) enables a planned reuse of software components into products within the same scope. The software product line engineering paradigm separates two processes; domain engineering (where the variability of the SPL is defined and realized) and application engineering (where specific software products are derived by reusing the variability of the SPL) (Pohl, Böckle, and Van Der Linden 2005).

The proactive strategy for the adoption of an SPL is traditionally regarded as the typical approach. Following this strategy, the assets of the SPL are developed prior to the derivation of any product (Krueger 2002). However, a recent survey reveals that only a minority of industrial SPLs are planned proactively, being the extractive approach more used (where existing products are re-engineered into an SPL) (Berger et al. 2013).

In particular, in model-based SPLs, the members of the SPL are specified in the form of models. However, in the context of a family of models, manually spotting the commonalities and variability among the models may become cumbersome and error prone, particularly as the number of models and its complexity increases.

There are several research efforts towards automating the formalization of the variability existing among products (Acher et al. 2013; Ziadi et al. 2012; Abbasi et al. 2014; She et al. 2011). However, those works are mainly based on Feature Models extraction and do not properly support variability formalization by means of the Common Variability Language (CVL). In addition, existing works (Zhang, Haugen, and Moller-Pedersen 2011; Rubin and Chechik 2012) are not designed with the evolution of the SPL on mind. The evolution of SPLs should be considered as the normal case, not as an anomaly (Dhungana et al. 2008).

This work presents Model Family to SPL, an approach to automate the variability formalization of a given family of models into an SPL. As output, the variability is made explicit in terms of CVL (Fleurey et al. 2009). The model commonalities are formalized as a base model and variabilities are specified as placements over the base model and replacements in a model library. In addition, the resulting SPL can be further evolved to include new products. In particular, our approach enables the automatic decomposition of new product models into model fragments that are incorporated to the model library.

We have validated the approach with our industrial partner (BSH), the largest manufacturer of home appliances in Europe. Their induction division has been producing induction hobs (under the brands of Bosch and Siemens among others) over the last 15 years. We have applied the presented approach to a set of their induction hobs models to build an SPL to generate the firmware for their products. In addition, we present the five evolution scenarios faced by our industrial partner when evolving the SPL to incorporate new product models.

The rest of the paper is structured as follows: next section introduces some background about the CVL and our industrial partner's domain. Section 1.3 presents our approach for extracting variability from a set of product models. In section 1.4 we present our experience applying the approach to our industrial partner's domain, focusing on the evolution scenarios encountered. Section 1.5 discusses related work. Finally we conclude the paper.

1.2 Background

This section presents the main concepts of the Domain Specific Language (DSL) used to specify Induction Hobs (hereinafter referred as IHs) and the CVL. Both, the Induction Hob Domain Specific Language (IHDSL) and CVL are the techniques which we use to describe the model-based SPL of our industrial partner.

1.2.1 Induction Hob Domain Specific Language (IHDSL)

The IHDSL metamodel used by our industrial partner is composed of 46 metaclasses, 74 references among them and more than 180 metaclass properties. However, in order to gain legibility and due to intellectual property rights concerns, in this paper we use a meaningful simplification of it (see top-left corner of Figure 1.1).

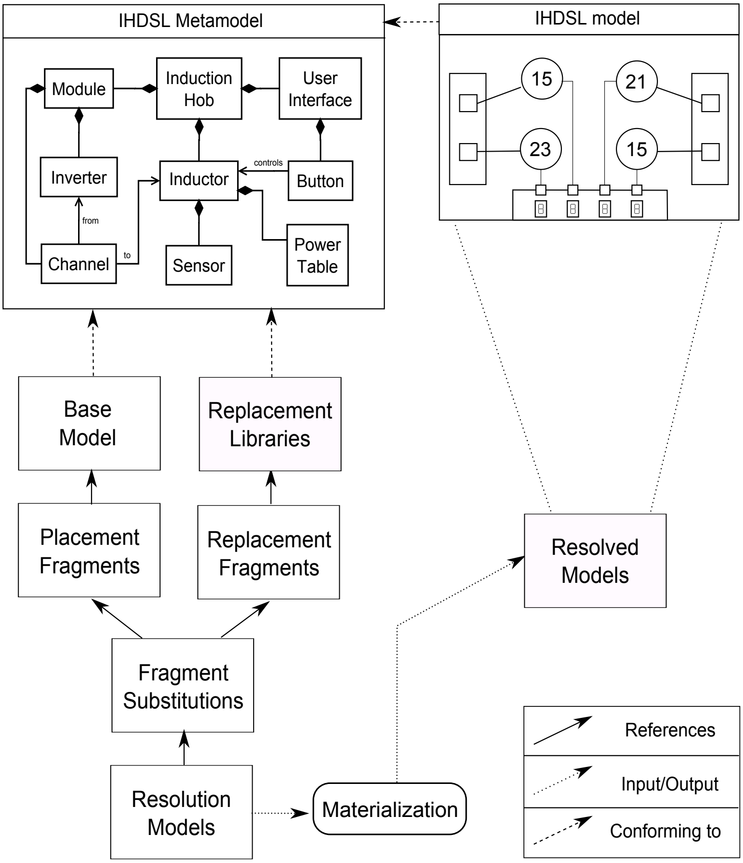


Figure 1.1: CVL Overview

Induction Hobs use electromagnetic induction phenomenon to cause the generation of heat on the cookware that is then transferred to the food. Induction hobs are composed of several elements, being the most important the inverter (where the energy is modulated) and the inductor (where the electromagnetic field is generated).

Top-right corner of Figure 1.1 shows the graphical representation of the IHDSL. The big rectangle represents the IH itself. It is composed of two power modules (vertical rectangles at both sides of the IH) and each of them holds two inverters (squares). Inverters are connected to the inductors (circles). The number inside each inductor represents the diameter of the inductor. The line that connects inverters and inductors represent the channel, which transfers energy from the inverter to the inductor. The user interface of an IH has buttons to configure the power level of each inductor. In top-right corner of Figure 1.1, the horizontal rectangle at the bottom of the IH represents the user interface. It has ports to connect each inductor with his button.

In order to gain legibility through the rest of the paper we will focus on the variability regarding the inductor. However, there are several parts of the induction hobs that are subject to variation such as the inverters and how are connected with the inductors. Our work with our industrial partner has covered the variability of the whole induction hob although only a subset is presented.

1.2.2 Common Variability Language (CVL)

CVL is a DSL for modeling variability in any model of any DSL based on Meta-Object Facility (MOF), an OMG‘s specification to define a universal metamodel for describing modeling languages. CVL defines variants of the base model by replacing parts of the base model by Model replacements found in a library. Figure 1.1 presents an overview of CVL.

The **base model** is a model described by a given DSL (here: IHDSL) that serves as the base for different variants defined over it. In CVL the elements of the base model 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**, a model described in the same DSL as the base model that will serve as a base to define alternatives for a placement. Each one of the alternatives for a placement is a **replacement fragment** (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** (hereinafter substitution). Each substitution references to a placement and a replacement and includes the information necessary to substitute the placement by the replacement. That is, 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 executed, the base model (with a placement substituted by a replacement) continues to conform to the same metamodel.

Each **resolution model** represents one variant of the base model. The resolution model references a set of substitutions that needs to be executed in order to create the variant. When a resolution model is materialized, produces a resolved model, which is a variant of the base model where the substitutions defined by the resolution model have been executed. For further details about the inner workings of CVL see (Fleurey et al. 2009).

1.3 Model Family to SPL

This section presents Model Family to SPL, our software process capable of turning the implicit variability existing among a given set of similar models into explicit variability. In particular, Model Family to SPL takes a product family modeled in any DSL (conforming to MOF) as input and generates a CVL based SPL where commonalities and variabilities among the model family are explicitly defined. That is, each of the models of the given model family are expressed in terms of CVL, resulting in an SPL capable of generating all the products from the given model family.

Figure 1.2 shows an example of execution of *Model Family to SPL*. Top part shows the input of the process, the model family. Bottom part shows the output of the process, an SPL formalized by CVL models. Middle part shows how the execution of the processes is performed. Product Family to SPL is composed of two sub-processes, **Select Base Model** and **Product Model to SPL**. *Select Base Model* analyses the given family of models and determines which one of them is more suitable to be the base model. Once the base model is selected, *Product Model to SPL* compares a product model from the model family with the base

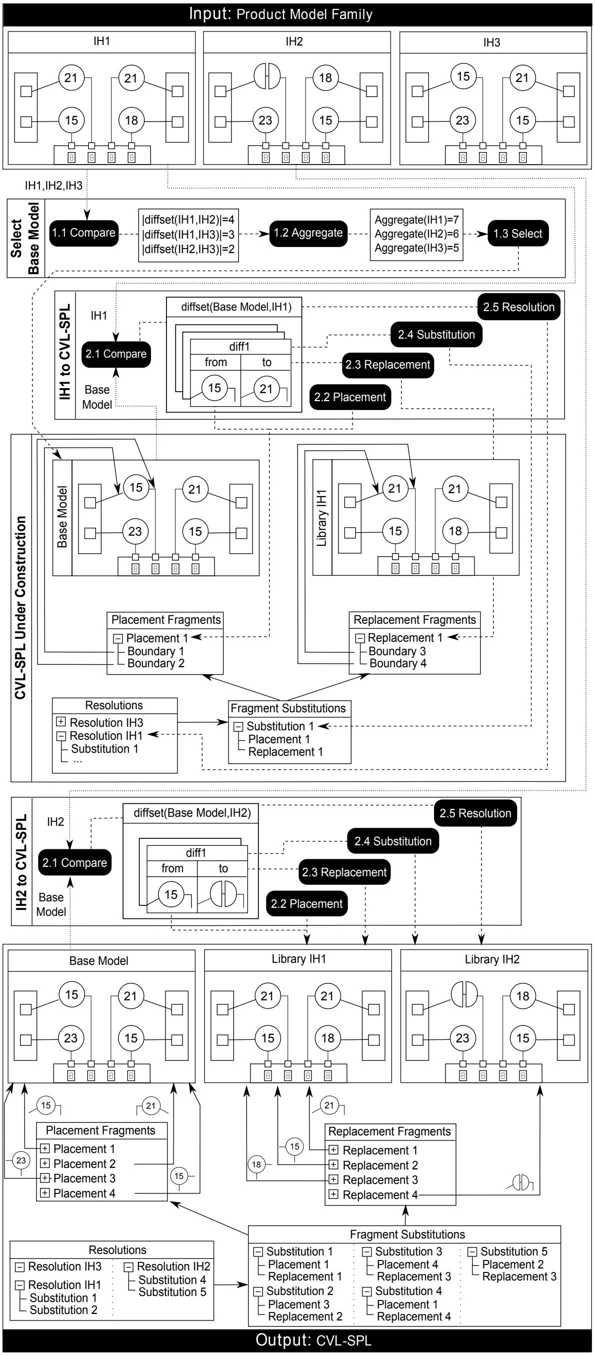


Figure 1.2: Model Family to SPL execution

model and updates CVL models to include the product into the variability definition. *Product Model to SPL* formalizes the variability of each given product model and incorporates it into the SPL

1.3.1 Select Base Model

The selection of the base model phase designates the base model that is used through the rest of the process. In this example we use the number of differences between the base model and the rest of the model family to determine the base model. Using the number of differences among the models produces simpler CVL models in terms of the number of substitutions needed to formalize each model. However, other values can be used to select the base model, the rest of the process can be executed no matter which base model is selected.

The first process executed as part of the Model Family to SPL is *Select Base Model.* Figure 1.2 shows an example of the execution of the process (top part). In addition, Figure 1.3 shows the state machine associated to the *Select Base Model* process (top part). Given a Product Model Family, the process *Select Base Model* takes the model family as input and proceeds as follows:

1.1 Compare. All the models from the model family (IH1, IH2 and IH3) are input into the compare operation. The models are paired two by two in all the possible combinations (the order doesn't matter) and then each pair is compared. The comparison is performed at element level, matching one element from first model with another from the second model. The process continues comparing pairs until there are no more pairs. The result is a set of differences between each pair of models processed. Figure 1.2 shows the result of the operation. For instance, the comparison between IH1 and IH2 produces a set of 4 diffs, because the four inductors of IH1 are different from the inductors of IH2.

1.2 Aggregate. The number of differences among each model and the rest of the models from the model family is added together. This is done for each of the models of the model family (IH1, IH2 and IH3). This aggregate value indicates the total number of differences

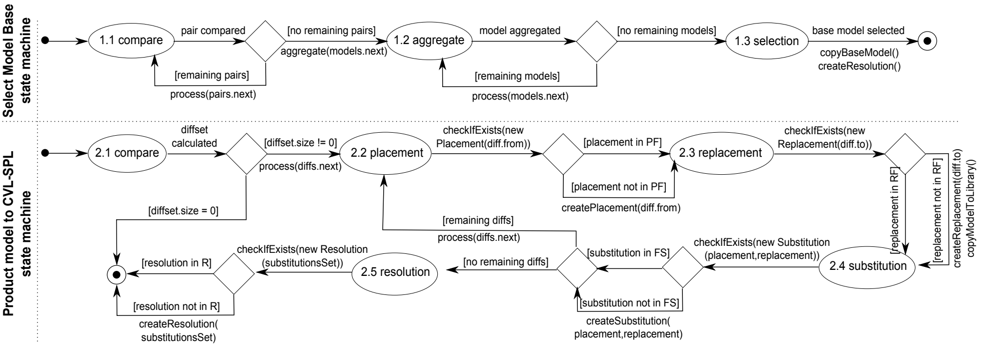


Figure 1.3: Select Base Model process and Product Model to SPL process state machines

among a given model and the rest of the product models. That is, the value indicates the number of differences that would need to be addressed if that particular model were the base model. For instance, if IH1 were the base model a total number of 7 differences would need to be addressed (4 differences with IH2 and 3 differences with IH3).

1.3 Select. When the aggregated values have been calculated for all the models, the model with the lowest value is designated as the base model. Therefore, it is included into the SPL, as it will be the base for all the products generated with the SPL. The model designated as base model (IH3 in this case) must be derivable from the SPL, therefore, a resolution model capable of generating the base model is created (Resolution IH3). As IH3 is the base model itself, there is no need of substitutions and Resolution IH3 is empty.

1.3.2 Product Model to SPL

After executing the first process (the base model has been designated), the second phase of the process starts, the population of the SPL. The population consists of executing the process *Product Model to SPL* for each of the product models of the input (except for the base model, that has been already included into the SPL).

The *Product Model to SPL* process, performs compare operations between each of the models from the given model family and the base model, using the same compare operation as in the previous process. However, this time we shall create and update the CVL models that define each of the differences among the model family as sets of placements, replacements, substitutions and resolutions.

Figure 1.2 shows an example of the execution of the *Product Model to SPL* process for IH1 (middle part, below *Select Base Model* process) and a snapshot of the SPL that is being constructed. In addition, Figure 1.3 shows the state machine for *Product Model to SPL* process (bottom part). Given a product model and a Base Model (designated by previous process), *Product Model to SPL* proceeds as follows:

2.1 Compare. The first model from the input model family (IH1) is compared with the base model. The result is a list of differences between the two models, *diffset(Base Model,IH1)*. Each difference has two elements, the *from* element references elements from the base model and the *to* element references elements from the other compared model (IH1 in this case). They reference the elements spotted as different by the compare operation. For instance, *diff1.from* element references the inductor of size 15 of the base model while *diff1.to* element references the inductor of size 21 of the IH1 model. It is important to notice that the difference not only holds the element that is different (inductor), but also the references involving that element (in this case references from the button and from the inverter).

2.2 Placement. The process checks if a placement holding exactly the same elements of *diff1.from* exists in the Placement Fragments model. As it does not exist, the process defines a placement over the base model (Placement 1). The references involving the differing element (the inductor) are defined as the boundaries of the placement (Boundary 1 and Boundary 2). If the placement is already defined in the Placement Fragments model it is not created again (see bottom of Figure 1.3).

2.3 Replacement. Once the placement is retrieved (created a new one or retrieving the existing one from the Placement Fragments model), the process continues with the replacement. Similarly as in previous step, the process checks if a replacement holding the information from *diff1.to* exists in the Replacement Fragments model. It does not exists, therefore it needs to be created, but this time will be defined over a model of the Replacements Library. To accomplish that, the model being processed (IH1) is copied into the fragments library and then a replacement is defined over it (Replacement 1). As with placement fragments, the references involving the differing element (the inductor) are defined as the boundaries of the replacement (Boundary 3 and Boundary 4). If the replacement is already defined in the Replacement Fragments model, there is no need to create a new one as indicated by the state machine (see bottom right part of Figure 1.3).

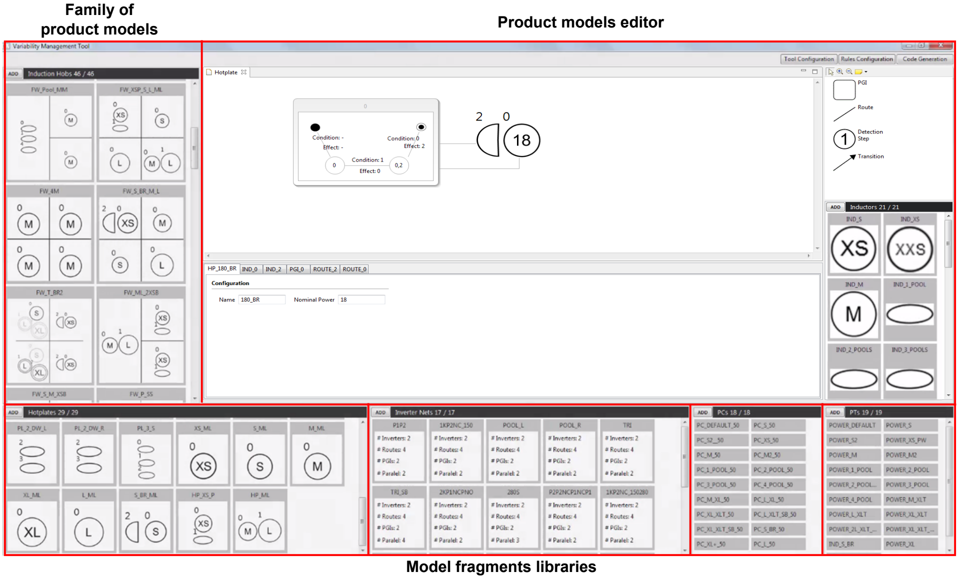


Figure 1.4: Resulting SPL for Induction Hobs domain

2.4 Substitution. Once the placement (Placement 1 from step 2.2) and the replacement (Replacement 1 from step 2.3) had been retrieved (creating them if necessary), the process is ready to create the substitution of the placement by the replacement. Similarly to previous steps, the process first checks if the substitution already exists in the Fragment Substitutions model. As the substitution does not exist, the process needs to create it. The substitution indicates that the Placement 1 can be substituted by the Replacement 1. As part of the definition of the substitution, links between the boundaries from the placement and the replacement are established. Therefore, when the fragment substitution is executed the elements can be updated properly and the model continues to conform to the metamodel. Similarly to previous step, if the substitution already exists there is no need to create it (see bottom of Figure 1.3).

At this point, the first difference (*diff1*) from the *diffset(Base Model,IH1)* has been processed. Now, the steps 2.2, 2.3 and 2.4 are performed for the rest of differences of the diffset. For each difference, a placement, a replacement and the proper substitution of the placement by the replacement are obtained (created or retrieved if already exists). The iterations for *diff2* and *diff3* are not shown in Figure 1.2.

2.5 Resolution. When all the differences from the diffset had been processed, the process is ready to create a resolution for the processed model (IH1). First, the process checks if the resolution already exists in the resolutions model. As the resolution does not exist, the process creates a new one (Resolution IH1). In this case, the process indicates that the resolution of IH1, involves the Substitution 1 (substitution of Placement 1 by Replacement 1) corresponding to the first diff processed. Similarly, substitutions for the rest of the differences of the diffset are included in this resolution.

This five-step process is repeated for all the models from the input (except for the base model). After executing *IH1 to SPL*, comes the execution of *IH2 to SPL*. The result is an SPL populated with all the models from the input family. Bottom part of Figure 1.2 shows the output of the *Model Family to SPL* operation. There is a base model and two library models conforming to the IHDSL. In addition, there are placements defined over the base model, and replacements defined over the library models. Moreover, substitutions are defined referencing placements and replacements;

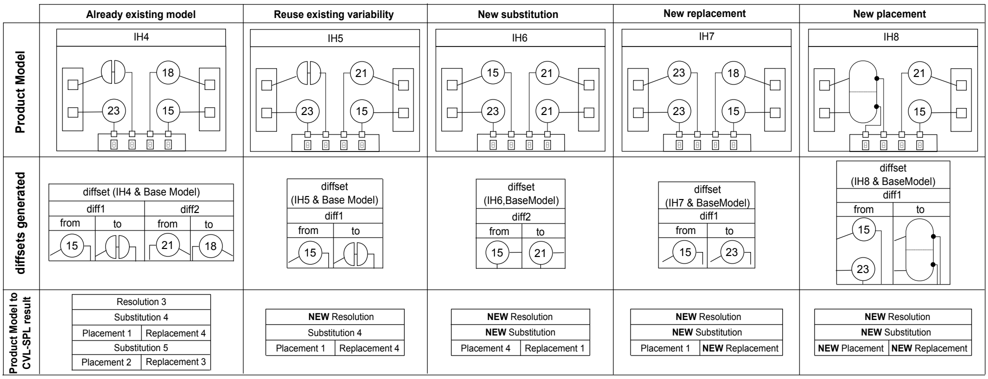


Figure 1.5: Five Scenarios of SPL evolution

resolutions that generate each of the models received as input have been created based on those substitutions.

With the above process we obtain a CVL-based SPL capable of generating exactly the same models provided as input of the process. However, the commonalities and variabilities among the products are now explicitly formalized in terms of CVL. In addition, the *Product Model to SPL* process presented can be used to further evolve the variability of the SPL, decomposing new products and expressing them in terms of CVL. Next section presents the application of the approach to our industrial partner and the set of evolution scenarios encountered.

1.4 Case Study: Induction Hobs

This section presents our experience building a Product Line from an existing set of products from our industrial partner (BSH group). This company is the largest manufacturer of home appliances in Europe and one of the leading companies in the sector worldwide. Their induction division has been producing induction hobs (the brand portfolio is composed by Bosch and Siemens among others) over the last 15 years.

In order to implement the approach, several technologies are involved. Specifically, CVL can be applied to MOF based models, so the approach is developed within the Eclipse environment using the Ecore implementation and the Eclipse Modeling Framework (EMF)[1]. The comparisons among models are implemented based on EMF-Compare[2], which is an Eclipse framework to compare instances of EMF models. To build the frontend of the SPL we have used the Graphical Modeling Project (GMP)[3], a framework that provides a set of generative components and runtime infrastructures for developing graphical editors based on EMF. Finally, to add variability management capabilities to the graphical editor we have integrated the CVL tool from Sintef (Fleurey et al. 2009), a CVL prototype implementation that can be integrated into editor.

The initial input of the approach is a set of 46 induction hob models, corresponding to products that are currently being sold or that will be launched to the market in the immediate future. The set of models were developed following a clone and own (Pham et al. 2009) approach, where each IH has been modeled modifying a copy of the most similar IH present in the collection. For instance, a modification includes taking some elements from other induction hobs and customize them (if necessary, sometimes the elements do not require further customization). Therefore, the variability present among the models has not been explicitly defined, resulting in a set of models with implicit variability among its members. With regard to the products complexity, each of the IH models is composed of more than 500 elements, including around 100 class elements on average.

Figure 1.4 presents the resulting SPL tool that makes use of the variability information obtained applying the *Product Model to SPL* process. Top left part presents the Induction Hobs that have already been derived from the SPL. The set of products is the same as the one used as input; however, those induction hobs have been expressed in terms of the reusable model fragments extracted through the *Product Model to SPL* process. Bottom part presents the libraries of model fragments, holding the 102 replacement model fragments obtained by the approach. When deriving new products, the model fragments presented by the libraries can be reused. Finally, top right part presents the editor area, where product models can be derived and customized.

[1] http://www.eclipse.org/modeling/emf/

[2] https://www.eclipse.org/emf/compare/index.html

[3] http://eclipse.org/modeling/gmp/

The tool contains the variability information extracted from the set of product models used as input. However, this information is extended when new product models are derived reusing existing model fragments (as the variability model needs to include the new product) and when new reusable model fragments are needed (the fragments need to be added to the model fragment libraries). For instance, one of our industrial partner engineer's creates a new empty model and populates it reusing elements from the library. Then, the engineer customizes some elements of the induction hob model using the editor and saves it. The *Product Model to SPL* process is automatically executed to include the new induction hob into the SPL, which can lead to an increment in the variability that is defined in the SPL or in the reusable model assets available to derivate further products.

Figure 1.5 presents five different examples that illustrates five different situations encountered when adding new models to the SPL. Each column presents one of the five examples. First row present the product model that is going to be added to the SPL. Second row shows the diffset generated when each model is compared with the Base Model (see bottom of Figure 1.2). Third row presents a summary of the changes that the application of *Product Model to SPL* produces over the CVL models. Next subsections present the five different scenarios.

1.4.1 Already existing model

First column is an example of the addition of a model that already exists in the SPL. The comparison between IH4 and the Base model produces a set of two differences (second row). When performing steps 2.2, 2.3 and 2.4, the placement, replacement and substitutions necessary to model diff1 already exists in the CVL models. Diff1 corresponds to already existing Substitution 4 (substitute Placement 1 by Replacement 4). Therefore, no placement, replacement or substitution is created for diff1. The same happens with diff2, that corresponds to Substitution 5 (substitute Placement 2 by Replacement 3). During the creation of the resolution model (step 2.5), the process detects that the resolution already exists in the SPL (Resolution 3 composed of Substitution 4 and Substitution 5). Therefore, no resolution is created as part of step 2.5.

When the step 2.5 does not involve the creation of a new resolution model (as in this scenario), denotes that the model being processed is already part of the SPL. The process *Product Model to SPL* automatically skips the inclusion of this model in order to avoid duplicates. By means of this scenario, we avoid the inclusion of redundancy into the SPL.

1.4.2 Model reusing existing variability

Second column is an example of the addition of a model that reuses the variability already defined in the SPL to generate a new product model. The comparison between IH5 and the Base model produces a set of one difference (second row). Execution of steps 2.2, 2.3 and 2.4 detects that diff1 corresponds to Substitution 4 (substitute Placement 1 by Replacement 4). During step 2.5 the resolution model does not exist in the SPL, therefore, a new resolution model that includes Substitution 4 is created.

When *Product Model to SPL* does not create any substitution means that already existing variability is being used to create a new product model. However, if the resolution model does not exist in the SPL, a new resolution including the substitutions identified for each diff is created. By means of this scenario, we have created a new product reusing existing variability.

1.4.3 Model requiring a new substitution

Third column shows the addition of a model that needs the creation of a new substitution in order to be included into the SPL. The comparison between IH6 and the Base model produces a set of only one difference (second row). Then, during step 2.2 a placement for diff1.from is identified (Placement 1). Similarly, during step 2.3 a replacement for diff1.to is identified (Replacement 4). However, during step 2.4 no existing substitution is identified, therefore a new one is created. Then, during step 2.5 a new resolution is created, holding the new substitution created in previous step.

Sometimes the placement, replacement and substitution for a given diff already exists in the SPL (as in previous scenario) while other times only the placement and replacement exists and a new substitution is created (as in this scenario). However, in both cases we are reusing already existing model fragments to create new product models. By means of this scenario we show how the existing variability is reused in the creation of new product models.

1.4.4 Model requiring a new replacement

Fourth column is an example of the addition of a model that requires the creation of a new replacement in order to be formalized and included into the SPL. The comparison between IH7 and the base model produces a set of one difference (second row). Step 2.2 determines that diff1.from correspond to the already existing Placement 1. By contrast, Step 2.3, determines that there is no replacement corresponding to diff1.to in the SPL models, therefore it is created. As a new replacement has been created in step 2.3, step 2.4 creates a new substitution of the Placement 1 by the just created replacement. Finally step 2.5 creates a new resolution model including the new substitution.

When the step 2.3 involves the creation of a new replacement, the next step 2.4 will always require the creation of a new substitution (as the substitution involves a new created placement, it cannot exist in the SPL). By means of this scenario, the variability defined in the SPL has been increased, including a new replacement that now is available for the construction of other models.

1.4.5 Model requiring a new placement

Fifth column is an example of the addition of a model that requires the creation of a new placement. The comparison between IH8 model and the base model returns a set of one difference (second row). Then, step 2.2 detects that there is no placement corresponding to diff1.from; therefore a new placement is defined over the base model. Then, in step 2.3 a new replacement defined by diff1.to is created in the SPL. As part of step 2.4, a new substitution (that substitutes the new placement by the new replacement) is created. Finally, the resolution model including the just created substitution is created as part of step 2.5.

If the step 2.2 involves the creation of a new placement, then, a new replacement (step 2.3) and a new substitution (step 2.4) will be also created. It is important to notice that during the inclusion of IH8 model into the SPL a new replacement has been created. This replacement overlaps with other existing replacements (Replacement 1 and Replacement 3), as it is defined over the same model elements as other existing placements. However, this situation does not poses a threat to the stability of the SPL models. Substitution in CVL can be restricted, to avoid situations where two overlapping placements try to be replaced. Therefore, it is safe to define overlapping placements as long as the restrictions among them are correctly defined.

1.5 Related Work

There are several research efforts in existing literature towards the automation of the variability formalization among a set of products. However, most of them are focused on generating Feature Models (FMs) and not address CVL particularities. For instance, (Acher et al. 2013) present an approach to reverse engineering and evolve architectural FMs. In particular, they focus on plugin-based systems, projecting variability and technical constraints of plugin dependencies into an architectural FM. In (Abbasi et al. 2014), the authors presents a reverse-engineering tool to extract variability data from web configurators and transform them into structured data (for instance, a feature model) in a semi-automated way. The tool incorporates a component that explores the configuration space simulating users' configuration actions in order to generate more variable data to be extracted.

Other research efforts rely on the source code of the products in order to extract the variability model. In (She et al. 2011) the authors present a tool-supported approach for reverse engineering FMs from different sources, such as Makefiles, preprocessor declarations, and documentation. They focus on identifying parents and combine logic formulas and descriptions as complementary sources of information. In addition, (Ziadi et al. 2012) propose an approach to identify features from the source code of products. They reduce the noise induced by spurious differences of various implementations of the same feature. Then, the process produce feature candidates that are manually pruned (to remove non-relevant candidates). However, these approaches rely on the source code level as input for the process, focusing in the generation of Feature Models. By contrast, our approach deals with the particularities of the CVL and is applied at model level.

In (Rubin and Chechik 2012), the authors propose a generic framework for mining legacy product lines and automating their refactoring to contemporary feature-oriented SPLE approaches. In (Martinez et al. 2014) the authors present MoVaC, an approach to identify and analyse commonalities and variability among a set of models, with the focus on the visualization of the results. In (Zhang, Haugen, and Moller-Pedersen 2011) the authors propose an approach to synthesize an SPL from the comparison of a set of models. The variability is extracted from the set of models and then a CVL model for the SPL is proposed. The approach is further refined in (Zhang, Haugen, and Moller-Pedersen 2012) to enable the inclusion of new models to the SPL. As output, a CVL model for the SPL is proposed to be manually enhanced. We further extend those works, automatically selecting a base model among the input models based on the metric desired. In addition, we have validated the approach building an SPL for an industrial environment, extracting the variability of a set of real induction hob models. Furthermore, we present the five different evolution scenarios encountered during the validation and how the approach handles them in order to evolve the variability of the SPL.

1.6 Conclusions

We have presented the *Model Family to SPL process*, capable of automating the formalization of the variability among a given set of similar product models. In addition, the generated SPL can be further extended in order to increase the variability specification. The presented approach has been tooled within the Eclipse environment using already existing technologies such as EMF Runtime, EMF Compare and GMP. Then, the approach has been validated with our industrial partner. Finally, we have presented the five different evolution scenarios encountered when evolving the variability specification by our industrial partner.

However, our current implementation has some limitations. For instance, the concrete syntax used to represent each of the elements from the library is not automatically produced. Therefore, some customization regarding the concrete syntax has been performed in order to present the resulting SPL tool. We plan to provide means for automating the generation of a graphical syntax following a generative approach (similar to the Graphical Modeling Project).

CVL materialization generates product models from resolution models. However, the graphical editor shows diagrams that need to be automatically generated for each resolved model. Therefore, the position of each graphical element needs to be calculated by custom layouts that automatically position each element in the correct place. Nevertheless, these limitations constitute our future work.

Bibliography

Abbasi, E.K. et al. (Feb. 2014). “Reverse engineering web configurators”. In: *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on*, pp. 264–273. DOI: 10.1109/CSMR-WCRE.2014.6747178 (cit. on pp. 31, 47).

Acher, Mathieu et al. (2013). “Extraction and evolution of architectural variability models in plugin-based systems”. English. In: *Software & Systems Modeling*, pp. 1–28. ISSN: 1619-1366. DOI: 10.1007/s10270-013-0364-2 (cit. on pp. 31, 47).

Berger, Thorsten et al. (2013). “A Survey of Variability Modeling in Industrial Practice”. In: *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*. VaMoS ’13. Pisa, Italy: ACM, 7:1–7:8. ISBN: 978-1-4503-1541-8. DOI: 10.1145/2430502.2430513 (cit. on p. 31).

Dhungana, D. et al. (2008). "Supporting Evolution in Model-Based Product Line Engineering". In: *Software Product Line Conference, 2008. SPLC '08. 12th International*, pp. 319–328. DOI: `10.1109/SPLC.2008.26` (cit. on p. 31).

Fleurey, Franck et al. (2009). "A Generic Language and Tool for Variability Modeling". In: *Technical Report SINTEF A13505* (cit. on pp. 31, 35, 43).

Krueger, CharlesW. (2002). "Easing the Transition to Software Mass Customization". English. In: *Software Product-Family Engineering*. Ed. by Frank van der Linden. Vol. 2290. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 282–293. ISBN: 978-3-540-43659-1. DOI: `10.1007/3-540-47833-7_25` (cit. on p. 31).

Martinez, Jabier et al. (2014). "Identifying and Visualising Commonality and Variability in Model Variants". English. In: *Modelling Foundations and Applications*. Ed. by Jordi Cabot and Julia Rubin. Vol. 8569. Lecture Notes in Computer Science. Springer International Publishing, pp. 117–131. ISBN: 978-3-319-09194-5. DOI: `10.1007/978-3-319-09195-2_8` (cit. on p. 48).

Pham, N.H. et al. (May 2009). "Complete and accurate clone detection in graph-based models". In: *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pp. 276–286. DOI: `10.1109/ICSE.2009.5070528` (cit. on p. 43).

Pohl, Klaus, Günter Böckle, and Frank Van Der Linden (2005). *Software product line engineering: foundations, principles, and techniques*. Springer (cit. on p. 31).

Rubin, Julia and Marsha Chechik (2012). "Combining Related Products into Product Lines". English. In: *Fundamental Approaches to Software Engineering*. Ed. by Juan de Lara and Andrea Zisman. Vol. 7212. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 285–300. ISBN: 978-3-642-28871-5. DOI: `10.1007/978-3-642-28872-2_20` (cit. on pp. 31, 48).

She, Steven et al. (2011). "Reverse Engineering Feature Models". In: *Proceedings of the 33rd International Conference on Software Engineering*. ICSE

'11. Waikiki, Honolulu, HI, USA: ACM, pp. 461–470. ISBN: 978-1-4503-0445-0. DOI: `10.1145/1985793.1985856` (cit. on pp. 31, 47).

Zhang, Xiaorui, O. Haugen, and B. Moller-Pedersen (Aug. 2011). "Model Comparison to Synthesize a Model-Driven Software Product Line". In: *Software Product Line Conference (SPLC), 2011 15th International*, pp. 90–99. DOI: `10.1109/SPLC.2011.24` (cit. on pp. 31, 48).

— (Dec. 2012). "Augmenting Product Lines". In: *Software Engineering Conference (APSEC), 2012 19th Asia-Pacific*. Vol. 1, pp. 766–771. DOI: `10.1109/APSEC.2012.76` (cit. on p. 48).

Ziadi, T. et al. (Mar. 2012). "Feature Identification from the Source Code of Product Variants". In: *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pp. 417–422. DOI: `10.1109/CSMR.2012.52` (cit. on pp. 31, 47).

Chapter 2

Towards Clone-And-Own Support: Locating Relevant Methods in Legacy Products

Clone-and-Own (CAO) is a common practice in families of software products consisting of reusing code from methods in legacy products in new developments. In industrial scenarios, CAO consumes high amounts of time and effort without guaranteeing good results. We propose a novel approach, Computer Assisted CAO (CACAO), that given the natural language requirements of a new product, and the legacy products from that family, ranks the legacy methods in the family for each of the new product requirements according to their relevancy to the new development. We evaluated our approach in the industrial domain of train control software. Without CACAO, software engineers tasked with the development of a new product had to manually review a total of 2200 methods in the family. Results show that CACAO can reduce the number of methods to be reviewed, and guide software engineers towards the identification of relevant legacy methods to be reused in the new product.

2.1 Introduction

Clone-and-Own (CAO) (Antkiewicz et al. 2014; Dubinsky et al. 2013; Pham et al. 2009; Rubin and Chechik 2013a; Rubin, Kirshin, et al. 2012) is a common practice in the development of new products in families of software products. It consists of reusing code from legacy products, modifying it to comply with the functionality particularities of the new product. Code reuse enables faster software development and easier tracking of projects, and helps maintain the code style consistent between products.

In the practice, CAO is carried out manually and relies on the knowledge that software developers have of the family. In industrial scenarios, families of software products tend to have a myriad of products with long and complex implementations, coded and maintained over long periods of time by different developers. In these scenarios, engineers tasked with new product developments often lack knowledge over the entirety of the products and their implementation details. Under these conditions, CAO is a process that consumes high amounts of time and effort without guaranteeing good results.

In this paper, we propose a novel approach, named Computer Assisted CAO (CACAO), that leverages Part-of-Speech tagging (POS tagging) (Hulth 2003) and adapts Latent Semantic Indexing (LSI) (Landauer, Foltz, and Laham 1998) to rank the relevancy of legacy products for a new development at the requirements level, and to locate their most significant methods for each of the new product requirements.

Given the natural language specifications of a new product in a family of software products, and the legacy products that belong to it, our approach detects which are the legacy products that are the closest to the new product in terms of requirements. In a second step, our approach searches the code of the closest legacy products for methods that are relevant for the new product requirements. As a result, our approach produces a code relevancy ranking for each of the requirements of the new product. Software engineers can benefit from the rankings to avoid the mentioned CAO issues.

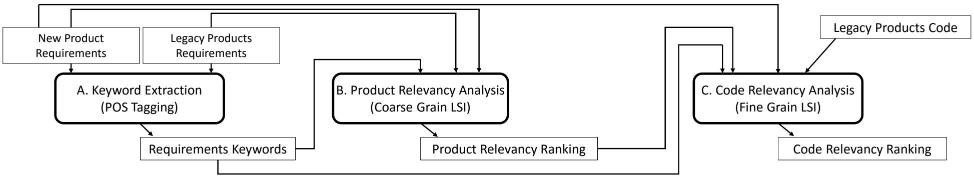

New Product Requirements
Legacy Products Requirements
Legacy Products Code
A. Keyword Extraction (POS Tagging)
B. Product Relevancy Analysis (Coarse Grain LSI)
C. Code Relevancy Analysis (Fine Grain LSI)
Requirements Keywords
Product Relevancy Ranking
Code Relevancy Ranking

Figure 2.1: Approach Overview

We evaluated our approach in the industrial domain of railway control software. Our industrial partner, Construcciones y Auxiliar de Ferrocarriles (CAF), provided a family of five software products used to control the trains they manufacture. In our evaluation, one product acts as a new product in the family, and the rest act as legacy products. The code of the product that acts as the new product is used as an oracle. We apply our approach to that scenario, and measure its performance in terms of recall and precision (Salton and McGill 1986) by comparing the results to the code of the oracle. These steps are followed five times, having all the products play the role of the oracle.

Results show that it is likely to find relevant code in the rankings. With CACAO, the amount of methods that software engineers review when developing a requirement for a new product is reduced: it is only needed to review a percentage of the original 2200 methods to build a new product.

The remainder of the paper is structured as follows: Section 2.2 presents our approach and shows how to apply it to a running example. Section 2.3 shows the evaluation of our work. Section 2.4 discusses the results of our work. Section 2.5 postulates the threats to the validity of our work. Section 2.6 comprehends the works related to this paper. Section 2.7 presents the conclusions of our work.

2.2 Approach

The goal of our approach is, given a set of natural language requirements for a new product, and the legacy products, to provide code relevancy rankings that enable software engineers to reduce the amount of methods they must review to develop the new product. To this extent, a series of steps are followed (see Figure 2.1):

A. First, the relevant keywords from the new product requirements and the legacy product requirements are extracted through extended POS Tagging techniques.

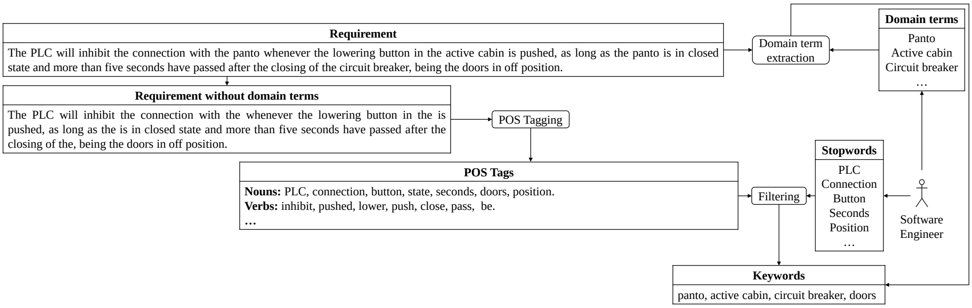
B. The second step of our approach performs a Coarse Grain LSI (CG-LSI) process to detect which of the legacy products are the closest to the new product in terms of requirements.

C. The third and last step of our approach is to perform a Fine Grain LSI (FG-LSI) process at the code level to detect which of the methods in the close legacy products are related to the new product requirements.

In the following pages, we detail the steps of our approach in the above order. To illustrate them, we use a running example from our industrial partner, CAF (Construcciones y Auxiliar de Ferrocarriles, at http://www.caf.net/en). CAF is a worldwide leader company in the railway industry. Since its foundation more than 100 years ago, they develop rail solutions such as high speed trains, regional and commuter trains, metros, trams and Light Rail Vehicles.

2.2.1 Keyword Extraction

The first step of our approach extracts keywords from the natural language requirements of the new product and the legacy products in the family. There are plenty of techniques that perform text mining and information retrieval from natural language requirements such as the ones in (Ferrari, Spagnolo, and Dell'Orletta 2013; Bakar, Kasirun, and Salleh 2015; Alves et al. 2008; Dumitru et al. 2011). The analysis of POS tags in search for nouns and nominal structures in documents has shown



The PLC will inhibit the connection with the panto whenever the lowering button in the active cabin is pushed, as long as the panto is in closed state and more than five seconds have passed after the closing of the circuit breaker, being the doors in off position.

The PLC will inhibit the connection with the whenever the lowering button in the is pushed, as long as the is in closed state and more than five seconds have passed after the closing of the, being the doors in off position.

Nouns: PLC, connection, button, state, seconds, doors, position.
Verbs: inhibit, pushed, lower, push, close, pass, be.
…

Figure 2.2: Keyword Extraction

promising results when extracting keywords from technical documents (Hulth 2003; F. Liu et al. 2009). The removal of stopwords (frequently occurring words meaningless to information retrieval) also helps produce more accurate results when mining data in documents (Saif et al. 2014; Lo, He, and Ounis 2005; Silva and Ribeiro 2003).

The combination of the analysis of POS tags and removal of stopwords is a frequent practice that our approach adopts to extract the most relevant keywords from the requirements documents. First, our approach searches for domain terms, provided by the software engineers, in the requirements. Then, the POS tags of the words that form the requirements, domain terms excluded, are analyzed. Afterwards, the words are filtered by their syntactic role in the sentences, and finally refined with a set of stopwords, also provided by the software engineers.

In Figure 2.2, a requirement from our running example is provided. This requirement describes part of the functionality of the pantograph of a train. The pantograph is the element that is used to harvest energy from the overhead wires installed in train lines.

First, the terms from the list of domain terms present in the requirement are subtracted from the requirement and introduced into the keywords list. Afterwards, the POS tags of the words that compose the requirement, domain terms excluded, are extracted. In Figure 2.2, the result of tagging the example requirement is shown. In the figure, it is possible to appreciate words like ’panto’ or ’doors’ as nouns, and ’inhibit’ or ’close’ as verbs. The rest of the words are omitted in the figure.

After the POS tagging, a filtering process takes place. Nouns are taken as keyword candidates due to their importance for keyword extraction Hulth 2003. The rest of the words are discarded. Nouns are filtered with a set of stopwords provided by the software engineers. The nouns that do not belong to the list of stopwords are added to the keywords list. Figure 2.2 shows a sample of the stopwords provided by a software engineer, as well as the final list of keywords extracted from the example requirement.

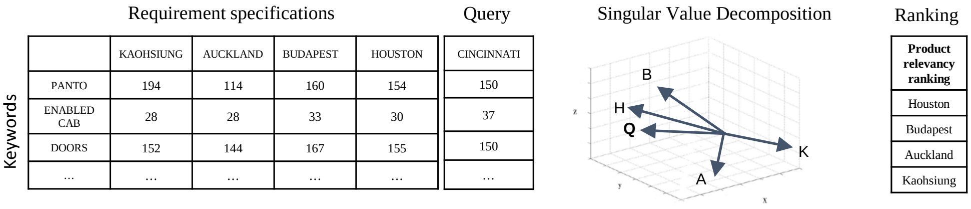
Keywords from all the requirements in all the documents are combined into a single set of terms, removing duplicates. The output of the first step of our approach is a set of terms with all the keywords extracted. These terms are used in the next steps of our approach.

2.2.2 Product Relevancy Analysis

In the second step of our approach, the keywords are used along with the new product and the legacy products requirements to perform a Coarse Grain LSI (CG-LSI). The aim of the CG-LSI process is to order the legacy products in a ranking that reflects their similarity to the new product development in terms of requirements.

Carrying out this step of our approach is relevant in industrial domains, where software families are conformed by a myriad of legacy products. In these scenarios, developers of a new product may lack knowledge of all the legacy products details. Through the product ranking, developers can appreciate whether the legacy products they know are relevant for the new development.

LSI (Rubin and Chechik 2013b) is an automatic mathematical/statistical technique that analyzes relationships between *queries* and *documents* (bodies of text). It constructs vector representations of both a user *query* and a corpus of text *documents* by encoding them as a *term-by-document co-occurrence matrix*, and analyzes the relationships between those vectors to get a similarity ranking between the *query* and the *documents*.



Requirement specifications / Query

Keywords	KAOHSIUNG	AUCKLAND	BUDAPEST	HOUSTON	CINCINNATI
PANTO	194	114	160	154	150
ENABLED CAB	28	28	33	30	37
DOORS	152	144	167	155	150
...	...	...	...	...	...

Ranking

Product relevancy ranking
Houston
Budapest
Auckland
Kaohsiung

Figure 2.3: Product Relevancy Analysis

In the second step of our approach, we adapted LSI to extract a ranking of the legacy products according to their similarity to the new product in terms of requirements. In our adapted CG-LSI, *terms* are the keywords extracted in the first step of our approach, *documents* are the legacy products requirements documents, and the *query* column is formed by the new product requirements document. Values of *term* occurrences in both the legacy product requirements documents and the new product requirements document are counted, and used to build the *term-by-document co-occurrence matrix.* The *documents* and the *query* are then transformed into vectors, and the relationships between the legacy product requirements documents and the new product requirements document are analyzed to extract the legacy product relevancy ranking.

Figure 2.3 shows the *term-by-document co-occurrence matrix* with the values associated to our running example, the vectors, and the resulting ranking. In the following paragraphs, an overview of the elements of the matrix is provided.

- Each row in the matrix stands for each unique keyword (*term*) extracted in the first step of our approach. In Figure 2.3, it is possible to appreciate a set of representative keywords in the domain such as 'PANTO' or 'DOORS' as the *terms* of each row.

- Each column in the matrix stands for the requirements document of each legacy product. In Figure 2.3, it is possible to appreciate the names of the legacy products in the columns such as 'KAOHSIUNG' or 'AUCKLAND', representing the requirements *documents* of those products.

- The final column stands for the *query*. In our approach, the *query* column stands for the requirements of the new product. In Figure 2.3, the name of the new product in the *query* column ('CINCINNATI') represents its requirements *document*.
- Each cell in the matrix contains the frequency with which the *term* of its row appears in the *document* denoted by its column. For instance, in Figure 2.3, the *term* 'PANTO' appears 114 times in the 'AUCKLAND' legacy product and 150 times in the 'CINCINNATI' new development.

We obtain vector representations of the *documents* and the *query* by normalizing and decomposing the *term-by-document co-occurrence matrix* using a matrix factorization technique called *singular value decomposition* (SVD) (Landauer, Foltz, and Laham 1998). SVD is a form of factor analysis, or more properly the mathematical generalization of which factor analysis is a special case. In SVD, a rectangular matrix is decomposed into the product of three other matrices. One component matrix describes the original row entities as vectors of derived orthogonal factor values, another describes the original column entities in the same way, and the third is a diagonal matrix containing scaling values such that when the three components are matrix-multiplied, the original matrix is reconstructed.

In Figure 2.3, a three-dimensional graph of the SVD is provided. On the graph, it is possible to appreciate each product, represented in the form a vector. The graph reflects the 'Houston' train vector as the closest to the new product vector, followed by the 'Budapest' train vector.

To measure the similarity degree between vectors, our approach calculates the cosine between the *query* vector and the *documents* vectors. Cosine values closer to one denote a higher degree of similarity, and cosine values closer to minus one denote a lower degree of similarity. Similarity increases as vectors point in the same general direction (as more *terms* are shared between *documents*). Having this measurement, our approach orders the legacy products according to their similarity degree to the new product in terms of requirements. The most similar legacy products are the ones that can be of the most relevance to the development process of the new product.

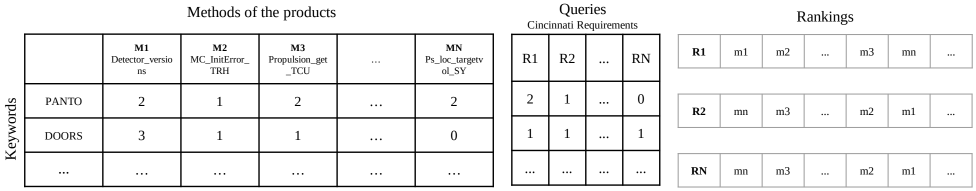
As the output of the second step of our approach, the product relevancy ranking (which can be seen in Figure 2.3) is produced according to the calculated similarity degrees. In our running example, our approach returns the legacy trains 'Houston' and 'Budapest' in the first and second position of the product relevancy ranking due to the cosines being '0.9243' and '0.8454', implying a high similarity degree with the new product in terms of requirements. On the opposite, the legacy train 'Kaohsiung' is returned in a latter position of the ranking due to its cosine being '-0.7836', a lower similarity degree.

The product relevancy ranking enables developers to decide whether to keep the legacy products familiar to them in the next step of CACAO, making a mixture between known and unknown products, or disregard them and only involve non-familiar products. Involving familiar products in the process is positive, since it is easier for software engineers to understand and reuse code known to them, but it should never enforce reusing code from non-relevant products.

2.2.3 Code Relevancy Analysis

In the third step of our approach, keywords are used along with the new product requirements, the product relevancy ranking, and the legacy products code to perform a Fine Grain LSI (FG-LSI) at the code level. The aim of the FG-LSI process is to order the methods of the legacy products in a ranking that reflects how similar they are to each of the new product requirements.

In the third step of our approach, we adapted LSI to extract a ranking of the methods in the relevant products that are of importance to the development of each new product requirement. In our adapted FG-LSI, *terms* are the keywords extracted in the first step of our approach, *documents* are the methods of the relevant legacy products, and there are several *query* columns, each of them a requirement of the new product development. Notice that in the third step of our approach, several instances of the *term-by-document co-occurrence matrix* are generated (one per query



	M1 Detector_versions	**M2** MC_InitError_TRH	**M3** Propulsion_get_TCU	...	**MN** Ps_loc_targetvol_SY	R1	R2	...	RN
PANTO	2	1	2	...	2	2	1	...	0
DOORS	3	1	1	...	0	1	1	...	1
...	...	...	...	...	...	...	...	...	...

R1	m1	m2	...	m3	mn	...
R2	mn	m3	...	m2	m1	...
RN	mn	m3	...	m2	m1	...

Figure 2.4: Code Relevancy Analysis

column). Values of *term* occurrences in both the methods and each of the requirements are counted, and used to build the matrices. The *documents* and the *queries* are transformed into vectors, and the relationships between the documents and each query are analyzed to extract a code relevancy ranking for each new product requirement.

For the sake of legibility, Figure 2.4 shows the LSI *term-by-document co-occurrence matrix* in a unified fashion (showing the values of the occurrences of the terms only once and grouping the queries to the right of the matrix). The figure also shows the values associated to this step of our running example and the resulting rankings. In the following paragraphs, an overview of the elements that a matrix contains is provided.

- Each row in the matrix stands for each of the unique keywords (*term*) extracted in the first step of our approach. In Figure 2.4, it is possible to appreciate a set of representative keywords in the domain such as 'PANTO' or 'DOORS' as the *terms* of each row.

- Each column in the matrix stands for each of the methods of the most relevant legacy products obtained in the previous step of our approach. A method *document* is composed by the name of the method, its variables, and the comments that appear in its body. External comments are not taken in account since we cannot ensure their belonging to a certain method. In Figure 2.4, columns M1 to MN represent the documents of those methods. Columns are labeled with method names, such as 'Detector_versions' or 'Propulsion_get_TCU'.

- In this step of our approach, there are several *query* columns. Each *query* column stands for each requirement of the new product. In

Figure 2.4, columns R1 to RN represent the requirements of the new 'CINCINNATI' train. The top part of Figure 2.2 on page 57 shows the R1 requirement of the 'CINCINNATI' train.

- 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 Figure 2.4, the *term* 'PANTO' appears twice in both M1 and R1.

We use the SVD technique presented in the second step of our approach to calculate the vectors of the *documents* and the *query* for each one of the matrices. The vectors are represented in graphs similar to the one in Figure 2.3, that, for space reasons, were omitted in the figure.

For each graph, our approach calculates the cosines between the *query* vector and the *document* vectors to measure the similarity degrees between them. Having the measurement of the similarity of the legacy product methods with each requirement, and reasoning that the most similar legacy products methods to a particular requirement are the ones that can be of the most relevance to its development, we provide an ordered list of the legacy products methods for each requirement according to their relevance in the new requirement development.

As the output of the third step of our approach, the code relevancy rankings for each requirement (which can be seen in Figure 2.4) are returned. In our running example, our approach returns the legacy methods 'm1' and 'm2' in the first and second position of the code relevancy ranking for the requirement 'R1' due to their cosines being '0.8743' and '0.6354', implying a high similarity degree between the code of those methods and the requirement. On the opposite, the legacy method 'mn' is returned in a latter position of the code relevancy ranking for the requirement 'R1' due to its cosine being '-0.7891', a lower similarity degree between code and requirement. This process is applied to all the requirements.

Software engineers in the company faced with the development of the new product can use these rankings to browse the most relevant methods for each requirement that they need to implement, avoiding the CAO issues.

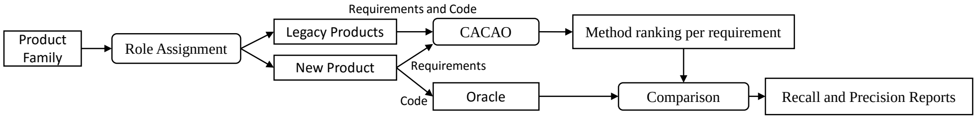


Figure 2.5: Evaluation Overview

2.3 Evaluation

This section evaluates our approach by applying it to a case study from our industrial partner, comprising a product family composed by five trains with an average of about 420 requirements each. Requirements have an average of around 50 words. Trains are coded by an average of 550 methods, with an approximate extension of 310 LOC each. Therefore, each train is coded in about 170.5 KLOC, and the family comprehends about 2750 methods that account for around 852.5 KLOC.

2.3.1 Evaluation Steps

Figure 6.4 shows the steps followed to evaluate our approach. We use the products in the roles of either legacy or new products to perform CACAO and get method rankings. Methods of the legacy products in the rankings are then compared with the real code of the new product, which acts as an oracle, to obtain precision and recall values that enable further analysis of the method rankings.

First, roles are assigned to products in the family. One product acts as the new product and the rest act as legacy products. The requirements and code of the products that act as legacy products, and the requirements of the product that acts as the new product are used to perform CACAO, while the code of the latter is kept apart to be used as an oracle.

CACAO performs the steps described in our approach (see Section 2.2) to provide a method ranking for each requirement of the new product. Notice the dimensions of the rankings extracted by CACAO. Products, on average, feature 420 requirements and 550 methods. For a new product, on average, 420 rankings are generated. Taking in account all the legacy products in our set as relevant products for the new development, each ranking orders about 2200 methods on average.

Then, the methods that compose the rankings are compared one by one with the code of the oracle. We perform the code comparison by carrying out a diff not only because version control software is really popular, and therefore there is a wide amount of tool support that calculates differences between two source codes available, but also because code comparison techniques have been used successfully for large scale systems (Li et al. 2006; Kamiya, Kusumoto, and Inoue 2002), proving the computational cost of the operation to be affordable for large documents like ours.

The effectiveness of information retrieval techniques is typically measured by recall and precision (Salton and McGill 1986). For a given query, recall is defined as the percentage of retrieved documents that are relevant to the total number of relevant documents, and precision is defined as the percentage of retrieved documents that are relevant to the total number of retrieved documents. All measures have values between 0 and 1 (Salman, Seriai, and Dony 2014). We calculate the recall and precision for every method by analyzing the results of the diff.

In our evaluation, the recall of a certain method represents the percentage of the oracle that is covered by the method. The recall of a method is calculated by counting the number of equal code lines between the method and the oracle code and measuring it against the total number of code lines of the oracle. The formula that represents the recall of a method is as follows:

$$Recall(Method) = \frac{LOC(Method \cap Oracle)}{LOC(Oracle)}$$

The precision of a certain method, on the other hand, represents the percentage of the method that appears inside the code of the oracle. The precision of a method is calculated by counting the number of equal code lines between the method and the oracle code and measuring it against the total number of code lines of the method. The formula that represents the precision of a method is as follows:

$$Precision(Method) = \frac{LOC(Method \cap Oracle)}{LOC(Method)}$$

In both formulas, the LOC function retrieves the number of lines of the element contained inside the parentheses, and the intersection between the method and the oracle represents the lines of code that are common to both the method and the oracle.

The steps of the evaluation described in the previous paragraphs are repeated as many times as the number of products in the product family, changing the product that acts as the new product in every iteration until every product in the product set has acted as the new product.

2.3.2 Implementation Details

The different steps carried out to perform and evaluate CACAO have been implemented using the following implementation frameworks:

- For the Keyword Extraction process (see section 2.2.1), the POS tags of the words that compose the requirements were extracted by using OpenNLP, a Natural Language Processing library developed by the Apache Software Foundation (at http://opennlp.apache.org/). This library provides a POS tagger implementation, along with POS tags models trained with machine learning techniques.
- To perform the necessary SVD in Latent Semantic Indexing (see Sections 2.2.2 and 2.2.3), EJML was used. EJML is a basic linear algebra package for Java (available at https://code.google.com/archive/p/efficient-java-matrix-library/). Along with other features, this library provides an implementation of SVD.
- In the evaluation of the results of the code rankings retrieved by CACAO against the oracle, the code diffs were carried out by leveraging the DiffUtils library. The DiffUtils library is a Java open source library which provides methods that enable us to perform the neces-

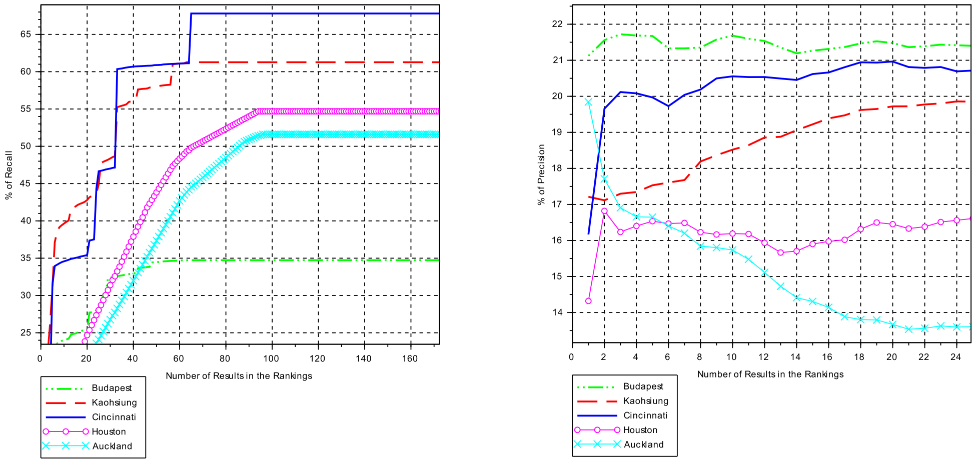


Figure 2.6: Recall and Precision results of the rankings

sary comparison operations between texts (at https://code.google.com/archiv diff-utils/).

For the evaluation of CACAO, we used a Lenovo E330 laptop, with a processor Intel(R) Core(TM) i5-3210M@2.5GHz with 16GB RAM and Windows 10 64-bit.

2.3.3 Results

Five iterations of the evaluation steps were run, with each of the five products playing the role of the new product and therefore being their code used as the oracle. Figure 2.6 shows two graphs that correspond to recall and precision results for CACAO when the 'Cincinnati' (solid line), 'Kaohsiung' (dashed line), 'Budapest' (discontinuous line), 'Houston' (dotted line), and 'Auckland' (crossed line) trains act as the new product.

For every requirement in the new product, CACAO generates one ranking. Each result in a ranking is composed by a method name, and the recall and precision values associated to that method. Results in each ranking are ordered by their relevance to the requirement development, determined by LSI.

Rankings can be shown with different numbers of results. For instance, a ranking showing its first result comprises the name of the most relevant method to the requirement, its recall value, and its precision value (i.e.: First result: m1 method, 2.76% recall, 63.41% precision). The same ranking, showing the first three results, comprises the names of three methods, their recall values, and their precision values (i.e.: First result: m1 method, 2.76% recall, 63.41% precision, Second result: m5 method, 3.52% recall, 72.49% precision, Third result: m8 method, 1.48% recall, 82.2% precision).

The left part of Figure 2.6 shows recall results of CACAO for the five new products. The horizontal axis represents the number of results shown in the rankings for all the requirements of the new product. The vertical axis represents the recall percentage, resulting from adding the recall of all the rankings generated. The formula for recall for one ranking, when k results are taken in account, is as follows:

$$Recall@k = \sum_{i=1}^{k} Recall(i) - C$$

Where C is calculated by adding the recall of the second and subsequent repetitions of the methods that have already appeared in the summation once.

For instance, a value of 37 in the horizontal axis, which returns a value of around the 26% total recall for 'Auckland' and around the 60% total recall for 'Kaohsiung' and 'Cincinnati', represents that when 37 results are shown in all the rankings, the recall of all the methods shown (not counting duplicate methods), adds up to around the 26% when the 'Auckland' train is the new product and up to around the 60% when either the 'Kaohsiung' train or the 'Cincinnati' train act as the new product.

By looking at recall results, it is possible to appreciate that the maximum recall (maximum percentage of the oracles that CACAO can cover) reaches up to the 67% for 'Cincinnati', 61% for 'Kaohsiung', 55% for 'Houston', 52% for 'Auckland', and 34% for 'Budapest', when each one is treated as the new product. In the cases of 'Kaohsiung' and 'Budapest', taking 60 results would suffice to fulfill the maximum recall, while in the case of 'Cincinnati' it would be necessary to increase the rankings size up to nearly 70 results, and more than 90 would be needed for 'Houston' and 'Auckland'. In the cases of 'Cincinnati', 'Budapest', and 'Kaohsiung', with rankings of 40 elements, around the 90% of the maximum recall would be achieved, while in the cases of 'Houston' and 'Auckland' about 80 results would be needed.

The right part of Figure 2.6 shows the precision results of CACAO for the five new trains. The horizontal axis represents the number of results shown in the rankings. The vertical axis represents the precision percentage associated to the results shown in the rankings, resulting from calculating the average precision of all the rankings, including duplicate methods. The formula for precision for one ranking, when k results are taken in account, is as follows:

$$Precision@k = \frac{\sum_{i=1}^{k} Precision(i)}{k}$$

For instance, a value of 12 in the horizontal axis, which is around the 15% precision for 'Auckland' and around the 21.5% precision for 'Budapest', represents that when 12 results are shown in all the rankings, the average precision of all the rankings shown revolts around the 15% when the 'Auckland' train is the new product and around the 21.5% when the 'Budapest' train is the new product.

Putting the focus on precision results, it can be appreciated that the maximum average precision (maximum average percentage of the methods that is present in the oracles) reaches up to around the 21.7% for 'Budapest', the 21% for 'Cincinnati', the 19.9% for 'Kaohsiung' and 'Auckland', and 16.9% for 'Houston', when each one is the new product. Rankings of around 5 positions would have precision values from around the 80% to almost 90% of the total precision in all cases except 'Auckland', where precision descends as the number of positions in the rankings augments. As more positions in the rankings are taken in account, values of precision become stable.

Data shows that it is likely to find relevant code in the rankings. CACAO results show that by reviewing a reduced percentage of the products presented, enough code can be found to cover a percentage of a new product. For instance, results show that by reviewing the first 37 positions of the rankings, relevant code can be found to cover between the 26% and the 60% of the new product.

We have pondered about the number of results in the rankings that software engineers need to look at in order to achieve useful code results, and we concluded that, in practice, it will not be necessary to review 37 methods per requirement to that extent. As pointed out by the second reviewer of our work, our metrics are affected due to oracles used in our evaluation being far from optimum. With an oracle that reflected all the possible code reuse, our recall and precision would improve, thus needing less ranking positions to achieve meaningful results. Further discussion about this fact can be found in the following section. Besides, it is reasonable to think that software engineers using CACAO will stop reviewing methods after finding out the code they need.

2.4 Discussion

By means of a Focus Group and semi-structured interviews with the software engineers, we compared their current CAO practice with the results of CACAO. The software of the five trains presented was developed by two different teams of software engineers. The two teams are geographically separated, but communicate through e-mail, periodic video-conferences, and weekly physical meetings. One of the teams (T1) developed the 'Houston' and 'Auckland' trains, while the other team (T2) developed the 'Budapest', 'Kaohsiung' and 'Cincinnati' trains.

We inquired the teams on whether they reviewed the code of the other team, and if so, on which percentage, when they develop a new product. T1 reported reviewing just a 5% of the code developed by T2, being that 5% mostly helper functions like signal delaying. T2 reported reviewing a 0% of the code developed by T1. However, the results of the CG-LSI performed by CACAO indicate that, given a train produced one team, the trains developed by the other team should be reviewed to obtain the maximum recall. In other words, given a new development by one team, the trains produced by the other team are relevant to perform CAO.

In Figure 2.3 it is possible to appreciate that for 'Cincinnati' (produced by T2), the most relevant train in terms of requirements is 'Houston' (produced by T1). Engineers confirmed that, with manual CAO, the code from 'Houston' was not used in the 'Cincinnati' development, and that it would never be used for a T2 development. With CACAO, engineers in T2 are suggested to use 'Houston' and its methods for their future products, even if they were not behind its development.

Through this kind of situations, we noticed that:

1. Since there is an independence between teams, methods from products developed by one team can be false positives in the rankings for oracles developed by the other team. The products of both teams may be similar regarding the terms used, but few lines of code will be actually shared between them since no code is reused in practice. These false positives appear in the method rankings, and present low recall and precision, affecting the metrics of our approach. Finding the proper way to filter out these false positives remains as future work.

2. We are lacking an ideal evaluation scenario. The ground truth is that the oracles used through our evaluation are software products coded through manual CAO. Due to the CAO limitations mentioned throughout this work, the code of the products used as oracles is far from perfection in the reuse aspect.

   Therefore, in our evaluation, we are comparing a version of code reuse that has been designed attending to the requirements specifications with oracles that are not built in this same manner but rather on a manual fashion and relying on human factors. It is not possible for developers to perfectly discern how much code can and should be reused for the development of a new product.

> Comparing the methods extracted by our approach with scenarios that lack the ideal conditions lowers our precision and recall. Should we encounter an oracle with the ideal code reuse conditions, where all the code from legacy products that could and should be reused has been reused and modified to some extent, values of recall and precision would increase as more lines of code would be shared between the methods and the oracle.

In addition, we analyzed why 'Budapest' presents much worse recall results than the rest of the trains presented in this study when it acts as the oracle. Inspecting the code, we could note that the variables are coded with a different naming convention than the one in the rest of the products. When evaluating CACAO, on the diff performed between the methods and the oracle, code deltas that represent modifications of the code are treated as completely different lines such as new lines or deleted lines. As the train variables are named different, recall levels lower. In the light of the results, code modifications should be analyzed instead of directly discarded.

To avoid this issue, we should consider using more Natural Language Processing techniques in future developments of CACAO. For instance, stemming (Porter 2001) should be used at some point of our approach. Stemming reduces words to their root. The objective is to unify words to avoid duplicity of terms. For example, 'coupling' will be stemmed to 'couple' or 'brakes' to 'brake'. This will allow us to retrieve concepts and keywords in an optimized fashion. As of today, applying this sort of techniques and analyzing their implications in our approach remains as future work.

2.5 Threats to validity

In this section we discuss some of the issues that might have affected the results of the evaluation and may limit the generalization of the results. We use the classification of threats to validity of (Runeson and Höst 2009; Wohlin et al. 2012) to acknowledge the limitations of our approach.

Construct validity: This aspect of validity reflects the extent to which the operational measures that are studied represent what the researchers have in mind. To minimize this risk, we measured the factors of recall and precision. These measures are widely accepted in the software engineering research community (Salton and McGill 1986; Salman, Seriai, and Dony 2014).

Internal validity: This aspect of validity is of concern when causal relations are examined. There is a risk that the factor being investigated may be affected by other neglected factors. The number of members in the family of trains may look small, but the products presented cover a wide range of railway types, from trams to medium-long distance trains. Furthermore, the products used in this study have been developed by different developer teams working for our industrial partner.

External validity: This aspect of validity is concerned with to what extent it is possible to generalize the finding, and to what extent the findings are of relevance for other cases. Software in the railway domain is representative of safety-critical systems like those present in the automotive domain or the aerospace domain. Nonetheless, CACAO should be applied to other domains before assuring its generalization.

Reliability: This aspect is concerned with to what extent the data and the analysis are dependent on the specific researcher. For our research, the data was recovered from trains chosen and provided by our industrial partner. The evaluation is performed by comparing the data with the trains themselves, acting as oracles.

2.6 Related Work

Approaches related to the one presented in this paper comprehend feature location techniques carried out at the code level. Typechef (Kästner et al. 2011) provides an infrastructure to locate the code associated to a given feature by means of analyzing the #ifdef directives. Trace analysis (Eisenberg and Volder 2005) is a run-time technique used to locate features. When the technique is executed, it produces traces indicating which parts of code have been executed.

Some approaches related to feature location use LSI to extract the code associated to a feature. Poshyvanyk et al. (Poshyvanyk et al. 2007) combine a scenario-based probabilistic ranking of events and information retrieval via LSI. Given a query formulated by the user to identify the feature and two sets of scenarios (one that exercises the feature and other that do not), their system ranks the program methods using LSI. They rank each executed method based on the frequency of its appearance in the trace. Liu et al. (D. Liu et al. 2007) combine information from an execution trace and from the comments and identifiers from the source code. They executed a single scenario, which exercises the desired feature, and all executed methods are identified based on the collected trace using LSI.

The prior techniques have been generally applied to searching the code of a feature that has to be extended or is involved in the fixing of a bug. Our approach extends the ideas of the previous works by involving the analysis of requirements and leveraging the fact that products form a family, instead of treating them as independent items. Unlike the previous works, our approach analyzes the requirements of the family of software products to determine which are the most relevant for reuse in the scenario of a new development, and later calculate rankings of the most relevant methods in the legacy products for the implementation of each requirement in the new product.

Feature location approaches in a product family such as the one presented in Xue, Xing, and Jarzabek 2012 center their efforts in finding the code that implements a feature between the different products by combining techniques such as FCA and LSI. In our approach, we are not interested in the best representation of a feature in the family, but in locating the most relevant methods that implement a requirement (regardless of whether it represents a feature, a fragment of a feature, or several features). Since engineers must review the proposed methods to decide what to reuse, our approach also differentiates from (Xue, Xing, and Jarzabek 2012) by introducing a step (Product Relevancy Analysis) where engineers decide over which products the location is made, balancing product relevancy and knowledge about the family: potentially, more code can be found on relevant products, but with a good level of knowledge of a product, it becomes easier for engineers to reuse code.

Other work (She et al. 2011) focuses on applying reverse engineering to the source code to obtain the variability model. In (Czarnecki and Wasowski 2007) the authors use propositional logic which describes the dependencies between features. In (Nadi et al. 2014) the authors combine Typechef techniques and propositional logic to extract conditions among a collection of features.

These works engage explicitly the variability of the legacy products, but do not indicate the most relevant methods in the legacy products for the development of each requirement in the new product, as our work does.

2.7 Conclusions

To keep pace with the increasing demand for custom-tailored software systems, companies often apply the Clone-and-Own practice, through which a new product in a software product family is built by copying and adapting code from other family products. Clone-and-Own is imperfect and in industrial scenarios, it can be a time and effort-consuming process without guaranteeing good results.

In this work, we show our approach, named Computer Assisted CAO (CACAO). Given a set of natural language requirements for a new product in a software product family, and the requirements and code of the legacy products, CACAO leverages Part-of-Speech tagging and Latent Semantic Indexing to rank the most relevant products to the new development at the requirements level first, and to locate the most relevant methods to each requirement of the new product in the second place. CACAO produces, for each requirement of the new product, a ranking of the most relevant methods in the family for the development of the requirement. Software engineers can use the rankings to avoid the mentioned CAO issues.

We have evaluated our approach on the railway domain with our industrial partner, Construcciones y Auxiliar de Ferrocarriles (CAF), who provided a family of five train control software products. The results of CACAO show that it is likely to find relevant code in the rankings. Furthermore, CACAO revealed products that were not considered to be reusable by the software engineers to be relevant for code reuse, as in the case of the 'Houston' train for the 'Cincinnati' train development. Finally, as future work, we plan to apply more Natural Language Processing techniques such as stemming to avoid the issues related to different naming conventions as seen in the 'Budapest' train, which achieved the lower recall values in our evaluation.

Bibliography

Alves, V. et al. (Sept. 2008). "An Exploratory Study of Information Retrieval Techniques in Domain Analysis". In: *Software Product Line Conference, 2008. SPLC '08. 12th International*, pp. 67–76. DOI: 10.1109/SPLC.2008.18 (cit. on p. 56).

Antkiewicz Michałand Ji, Wenbin et al. (2014). "Flexible Product Line Engineering with a Virtual Platform". In: *Companion Proceedings of the 36th International Conference on Software Engineering.* ICSE Companion 2014. Hyderabad, India: ACM, pp. 532–535. ISBN: 978-1-4503-2768-8. DOI: 10.1145/2591062.2591126 (cit. on p. 54).

Bakar, Noor Hasrina, Zarinah M. Kasirun, and Norsaremah Salleh (Aug. 2015). "Feature Extraction Approaches from Natural Language Requirements for Reuse in Software Product Lines". In: *J. Syst. Softw.* 106.C, pp. 132–149. ISSN: 0164-1212. DOI: 10.1016/j.jss.2015.05.006 (cit. on p. 56).

Czarnecki, Krzysztof and Andrzej Wasowski (2007). "Feature Diagrams and Logics: There and Back Again". In: *Software Product Lines, 11th International Conference, SPLC, Kyoto, Japan, September 10-14, 2007, Proceedings.* IEEE Computer Society. DOI: 10.1109/SPLINE.2007.24 (cit. on p. 75).

Dubinsky, Yael et al. (2013). "An exploratory study of cloning in industrial software product lines". In: *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on.* IEEE, pp. 25–34 (cit. on p. 54).

Dumitru, Horatiu et al. (2011). "On-demand Feature Recommendations Derived from Mining Public Product Descriptions". In: *Proceedings of the 33rd International Conference on Software Engineering.* ICSE '11. Waikiki, Honolulu, HI, USA: ACM, pp. 181–190. ISBN: 978-1-4503-0445-0. DOI: 10.1145/1985793.1985819 (cit. on p. 56).

Eisenberg, Andrew David and Kris De Volder (2005). "Dynamic Feature Traces: Finding Features in Unfamiliar Code". In: *21st IEEE International Conference on Software Maintenance (ICSM), 25-30 September 2005, Budapest, Hungary.* IEEE Computer Society, pp. 337–346. ISBN: 0-7695-2368-4. DOI: 10.1109/ICSM.2005.42 (cit. on p. 73).

Ferrari, Alessio, Giorgio O. Spagnolo, and Felice Dell'Orletta (2013). "Mining Commonalities and Variabilities from Natural Language Documents". In: *Proceedings of the 17th International Software Product Line Conference.* SPLC '13. Tokyo, Japan: ACM, pp. 116–120. ISBN: 978-1-4503-1968-3. DOI: 10.1145/2491627.2491634 (cit. on p. 56).

Hulth, Anette (2003). "Improved automatic keyword extraction given more linguistic knowledge". In: *Proceedings of the 2003 conference on Empirical methods in natural language processing.* Association for Computational Linguistics, pp. 216–223 (cit. on pp. 54, 57, 58).

Kamiya, Toshihiro, Shinji Kusumoto, and Katsuro Inoue (2002). "CCFinder: a multilinguistic token-based code clone detection system for large scale source code". In: *Software Engineering, IEEE Transactions on* 28.7, pp. 654–670 (cit. on p. 65).

Kästner, Christian et al. (2011). "Variability-aware parsing in the presence of lexical macros and conditional compilation". In: *Proceedings of the 26th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2011.* Ed. by Cristina Videira Lopes and Kathleen Fisher. ACM. ISBN: 978-1-4503-0940-0 (cit. on p. 73).

Landauer, Thomas K, Peter W Foltz, and Darrell Laham (1998). "An introduction to latent semantic analysis". In: *Discourse processes* 25.2-3, pp. 259–284 (cit. on pp. 54, 60).

Li, Zhenmin et al. (2006). “CP-Miner: Finding copy-paste and related bugs in large-scale software code”. In: *Software Engineering, IEEE Transactions on* 32.3, pp. 176–192 (cit. on p. 65).

Liu, Dapeng et al. (2007). “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. Atlanta, Georgia, USA: ACM, pp. 234–243. ISBN: 978-1-59593-882-4. DOI: 10.1145/1321631.1321667 (cit. on p. 74).

Liu, Feifan et al. (2009). “Unsupervised approaches for automatic keyword extraction using meeting transcripts”. In: *Proceedings of human language technologies: The 2009 annual conference of the North American chapter of the association for computational linguistics.* Association for Computational Linguistics, pp. 620–628 (cit. on p. 57).

Lo, Rachel Tsz-Wai, Ben He, and Iadh Ounis (2005). “Automatically building a stopword list for an information retrieval system”. In: *Journal on Digital Information Management: Special Issue on the 5th Dutch-Belgian Information Retrieval Workshop (DIR).* Vol. 5. Citeseer, pp. 17–24 (cit. on p. 57).

Nadi, Sarah et al. (2014). “Mining configuration constraints: static analyses and empirical results”. In: *36th International Conference on Software Engineering, ICSE 14, Hyderabad, India - May 31 - June 07, 2014.* Ed. by Pankaj Jalote, Lionel C Briand, and André van der Hoek. ACM, pp. 140–151. ISBN: 978-1-4503-2756-5. DOI: 10.1145/2568225.2568283 (cit. on p. 75).

Pham, Nam H et al. (2009). “Complete and accurate clone detection in graph-based models”. In: *Proceedings of the 31st International Conference on Software Engineering.* IEEE Computer Society, pp. 276–286 (cit. on p. 54).

Porter, MF (Oct. 2001). *Snowball: A language for stemming algorithms* (cit. on p. 72).

Poshyvanyk, Denys et al. (2007). “Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval”.

In: *IEEE Trans. Software Eng.* DOI: 10.1109/TSE.2007.1016 (cit. on p. 74).

Rubin, Julia and Marsha Chechik (2013a). “A Framework for Managing Cloned Product Variants”. In: *Proceedings of the 2013 International Conference on Software Engineering.* ICSE ’13. San Francisco, CA, USA: IEEE Press, pp. 1233–1236. ISBN: 978-1-4673-3076-3 (cit. on p. 54).

— (2013b). “A survey of feature location techniques”. In: *Domain Engineering.* Springer, pp. 29–58 (cit. on p. 58).

Rubin, Julia, Andrei Kirshin, et al. (2012). “Managing Forked Product Variants”. In: *Proceedings of the 16th International Software Product Line Conference - Volume 1.* SPLC ’12. Salvador, Brazil: ACM, pp. 156–160. ISBN: 978-1-4503-1094-9. DOI: 10.1145/2362536.2362558 (cit. on p. 54).

Runeson, Per and Martin Höst (2009). “Guidelines for conducting and reporting case study research in software engineering”. In: *Empirical software engineering* 14.2, pp. 131–164 (cit. on p. 72).

Saif, Hassan et al. (2014). “On stopwords, filtering and data sparsity for sentiment analysis of Twitter”. In: *LREC 2014, Ninth International Conference on Language Resources and Evaluation. Proceedings.* Pp. 810–817 (cit. on p. 57).

Salman, Hamzeh Eyal, Abdelhak Seriai, and Christophe Dony (2014). “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 (cit. on pp. 65, 73).

Salton, Gerard and Michael J. McGill (1986). *Introduction to Modern Information Retrieval.* New York, NY, USA: McGraw-Hill, Inc. ISBN: 0070544840 (cit. on pp. 55, 65, 73).

She, Steven et al. (2011). “Reverse engineering feature models”. In: *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011.* Ed. by Richard N Taylor, Harald C Gall, and Nenad Medvidovic. ACM. ISBN: 978-1-4503-0445-0. DOI: 10.1145/1985793.1985856 (cit. on p. 75).

Silva, Catarina and Bemardete Ribeiro (2003). "The importance of stop word removal on recall values in text categorization". In: *Neural Networks, 2003. Proceedings of the International Joint Conference on.* Vol. 3. IEEE, pp. 1661–1666 (cit. on p. 57).

Wohlin, Claes et al. (2012). *Experimentation in Software Engineering.* Springer Science & Business Media (cit. on p. 72).

Xue, Yinxing, Zhenchang Xing, and Stan Jarzabek (2012). "Feature Location in a Collection of Product Variants". In: *19th Working Conference on Reverse Engineering, WCRE 2012, Kingston, ON, Canada, October 15-18, 2012*, pp. 145–154. DOI: 10.1109/WCRE.2012.24 (cit. on p. 74).

Chapter 3

Leveraging Feature Location to Extract the Clone-and-Own Relationships of a Family of Software Products

Feature location is concerned with identifying software artifacts associated with a program functionality (features). This paper presents a novel approach that combines feature location at the model level with code comparison at the code level to extract Clone-and-Own Relationships from a family of software products. The aim of our work is to understand the different Clone-and-Own Relationships and to take advantage of them in order to improve the way features are reused. We have evaluated our work by applying our approach to two families of software products of industrial dimensions. The code of one of the families is implemented manually by software engineers from the models that specify the software, while the code of the other family is implemented automatically by a code generation tool.

The results show that our approach is able to extract relationships between features such as Reimplemented, Modificated, Adapted, Unaltered, and Ghost Features, thus providing insight into understanding the Clone-and-Own relationships of a family of software products. Furthermore, we suggest how to use these relationships to improve the way features are reused.

3.1 Introduction

Feature location is concerned with identifying software artifacts associated with a program functionality (features). Feature location is one of the most important and common activities performed by developers during software maintenance and evolution (Dit et al. 2013). Most of the approaches carry out feature location at the code level (Dit et al. 2013),(Eaddy et al. 2008),(Czarnecki and Wasowski 2007), but in recent years feature location at the model level is gaining momentum (Font, Ballarin, et al. 2015),(Rubin and Chechik 2012),(Martinez et al. 2015).

This paper presents the first approach that combines the recent techniques on feature location at the model level with code comparison at the code level. We combine both to extract Clone-and-Own Relationships from a family of software products where the software has been specified through models, and implemented either in a manual or in an automatic way. The extracted Clone-and-Own Relationships reflect how features have been reused throughout the development of the family of software products.

In order to combine both techniques, we used the information that the techniques on feature location provide to develop an algorithm that isolates features at the model level. Then, our approach uses that information to guide code comparisons at the code level. This enables us to isolate features at the code level and retrieve their source code. Finally, we make one-to-one comparisons of the source code of a feature isolated in a product with the source codes of the different isolations of the same feature in other products.

We have evaluated our approach in the industrial domain of Induction Hobs (IH) over two families of IH products. On one of them, the firmware code of the products was implemented manually from the models. On the other, the firmware code of the products was implemented in an automatic way.

The results show that it has been possible to identify several different Clone-and-Own Relationships between features such as Reimplemented, Modified,Adap-ted, Unaltered, and Ghost Features. These relationships

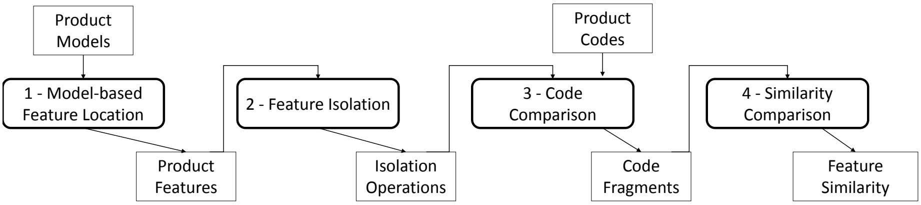


Figure 3.1: Stages of the Approach

are then used to suggest improvements on how features are reused. In the case of automatic implementation, extracted relationships are used to analyze whether it is necessary to carry out changes over the model-to-code transformation. In the case of manual implementation, extracted relationships are used to detect reuse impediments, to analyze cost-benefit and to detect opportunities to improve the reuse maturity.

The rest of the paper is structured as follows: Section 2 presents our approach and shows how to apply our approach to a simple example. Section 3 shows the evaluation of our work. Section 4 comprehends the work related to this paper. Section 5 summarizes the conclusions of our work.

3.2 Clone-and-Own Extraction Approach

The aim of our approach is to extract Clone-and-Own Relationships that enable us to understand and improve how features are reused among the products. The input of our approach is a family of software products where the software has been specified through models. The models are translated into code by humans or in an automatic way using a model-to-text transformation (Selic 2003). Our Clone-and-Own Extraction approach builds up on feature location at the model level and code comparisons. The main stages of our approach are: Model-based feature location, Feature Isolation, Code Comparison and Similarity Comparison. Fig. 4.3 depicts the inputs and outputs of these stages, which are described in the following subsections.

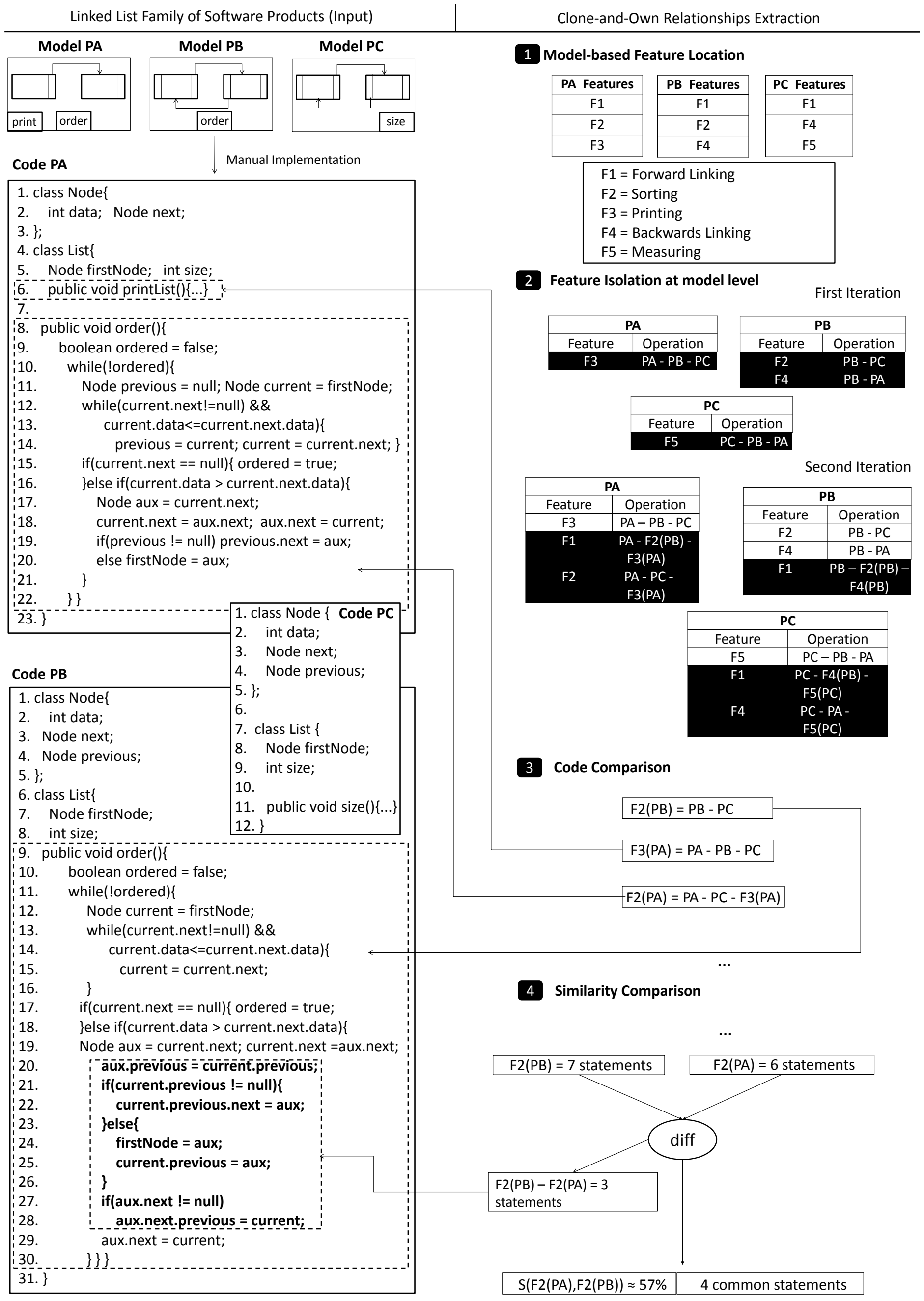
We use a running example in order to illustrate our approach. The Linked List Example is based on a family of software products where the variability is not formalized. The products have associated models, from which the code of the products has been manually implemented by a human (see left side of Fig. 4.4). The products are lists, which can be singly or doubly linked lists. Each list has a different combination of added functionality: sorting functionality (using the bubble method), functionality that enables calculating the number of elements of the list, and functionality that prints the elements of the list.

3.2.1 Model-based Feature Location

The first stage of our approach extracts the features from the products at the model level by using already existing techniques that identify features given a set of models. Feature location consists of identifying a fragment in the source code or software model that corresponds to a specific functionality. It is one of the most frequent maintenance activities undertaken by developers because it is a part of the incremental change process (Dit et al. 2013).

There are several research efforts in existing literature towards feature location from a set of models (Zhang, Haugen, and Møller-Pedersen 2011), (Martinez et al. 2015), (Rubin and Chechik 2012). For this stage we have adopted Conceptualized Model Patterns to feature location (hereinafter CMP-FL) (Font, Arcega, et al. 2015), which identify model patterns by human-in-the-loop (domain experts and application engineers become part of the decision-making process) and conceptualize the extracted patterns as reusable model fragments. We have adopted CMP-FL because the authors show CMP-FL improves the results obtained with previous approaches, providing features that are more recognizable by the engineers.

In CMP-FL, the elements that differ between the product models are extracted as alternatives for a feature. The elements that do not have a counterpart in the rest of the models are extracted as optional features. As a result, the models will be divided into reusable model fragments. Each of the reusable fragments will correspond with one of the features



Code PA

```
class Node{
    int data;   Node next;
};
class List{
    Node firstNode;   int size;
    public void printList(){...}

public void order(){
    boolean ordered = false;
    while(!ordered){
        Node previous = null; Node current = firstNode;
        while(current.next!=null) &&
            current.data<=current.next.data){
              previous = current; current = current.next; }
        if(current.next == null){ ordered = true;
        }else if(current.data > current.next.data){
            Node aux = current.next;
            current.next = aux.next;  aux.next = current;
            if(previous != null) previous.next = aux;
            else firstNode = aux;
        }
    } }
}
```

Code PC

```
class Node {
    int data;
    Node next;
    Node previous;
};

class List {
    Node firstNode;
    int size;

    public void size(){...}
}
```

Code PB

```
class Node{
    int data;
    Node next;
    Node previous;
};
class List{
    Node firstNode;
    int size;
public void order(){
    boolean ordered = false;
    while(!ordered){
        Node current = firstNode;
        while(current.next!=null) &&
            current.data<=current.next.data){
              current = current.next;
        }
        if(current.next == null){ ordered = true;
        }else if(current.data > current.next.data){
        Node aux = current.next; current.next =aux.next;
            aux.previous = current.previous;
            if(current.previous != null){
              current.previous.next = aux;
            }else{
              firstNode = aux;
              current.previous = aux;
            }
            if(aux.next != null)
              aux.next.previous = current;
            aux.next = current;
        } } }
}
```

PA Features
F1
F2
F3

PB Features
F1
F2
F4

PC Features
F1
F4
F5

F1 = Forward Linking
F2 = Sorting
F3 = Printing
F4 = Backwards Linking
F5 = Measuring

First Iteration

PA	
Feature	Operation
F3	PA - PB - PC

PB	
Feature	Operation
F2	PB - PC
F4	PB - PA

PC	
Feature	Operation
F5	PC - PB - PA

Second Iteration

PA	
Feature	Operation
F3	PA – PB - PC
F1	PA - F2(PB) - F3(PA)
F2	PA - PC - F3(PA)

PB	
Feature	Operation
F2	PB - PC
F4	PB - PA
F1	PB – F2(PB) – F4(PB)

PC	
Feature	Operation
F5	PC – PB - PA
F1	PC - F4(PB) - F5(PC)
F4	PC - PA - F5(PC)

F2(PB) = PB - PC
F3(PA) = PA - PB - PC
F2(PA) = PA - PC - F3(PA)

F2(PB) = 7 statements
F2(PA) = 6 statements
F2(PB) – F2(PA) = 3 statements
S(F2(PA),F2(PB)) ≈ 57%
4 common statements

Figure 3.2: Clone-and-Own Relationships Extraction applied to the Linked List Example

of the family of software products. The output of our first stage is a list for each product, that contains the features of the product which have been located at the model level by CMP-FL.

The Linked List Example (see 1 Model-based Feature Location of Fig. 2) tags the products with the located features. In the figure, the products, their features, and the names associated with the features are shown. In this example, five features are identified in the product family.

Current techniques used to locate features at the model level (Zhang, Haugen, and Møller-Pedersen 2011), (Martinez et al. 2015), (Rubin and Chechik 2012) (Font, Arcega, et al. 2015) do not provide meaningful names, only synthetic names (F1, F2, etc). We have decided to add more meaningful names to the features in order to improve understanding of the example: F1, (Forward Linking), F2 (Sorting), F3 (Printing), F4 (Backwards Linking) and F5 (Measuring).

In the first product (PA), features F1, F2 and F3 have been detected. In the second product (PB), features F1, F2, and F4 have been detected. Finally, in the third product (PC), features F1, F4, and F5 have been detected.

Notice that some of the features are present in more than one product. For instance, F2 is present in both product PA and product PB. In order to avoid ambiguity in feature names through this example, a feature FN that belongs to a product PX will be referred to as FN(PX).

3.2.2 Feature Isolation

This stage performs subtractions between the different products at the model level to identify the features that can be potentially isolated in code. We developed an algorithm that performs the second stage. The algorithm's input is a list of the existing products and their features. The result of the algorithm is the list of the features that can be isolated at the model level, accompanied by one operation per feature which expresses the code subtractions that need to be carried out between products in order to isolate the mentioned feature. The implementation of the algorithm is described as follows:

- The algorithm creates an empty list to store the features that it is able to isolate.
- For each feature (FN) of every product (PX), the algorithm calculates the Complementary Feature Set (CFS). A CFS is a product, combination of products, or combination between products plus already isolated features which contains all the features in PX except for FN. A CFS is valid even if it contains features that are not present in PX. Subtracting the found CFS to PX results in isolating FN. The isolation operation becomes FN(PX) = PX - CFS (e.g.: F1(P7) = P7 - P6 - F3(P4)).
- The isolated features and their isolation operations are added to the list. The addition of new features to the list of isolated features enables for new CFS, hence new feature isolations, so we make iterations while new isolated features are added to the list.

The first iteration of the algorithm will include into the list those features that can be isolated by a CFS composed only of a product or combination of products. Isolation operations found in the first iteration constitute the base cases of our algorithm. Following iterations will use combinations between products plus already isolated features to calculate the CFS. Isolation operations found this way constitute the recursive cases of our algorithm.

The Linked List Example (see 2 Feature Isolation at model level of Fig. 2) shows the application of our feature isolation algorithm as follows.

- **First Iteration:** For all the features in PA, the feature isolation algorithm searches for the CFS that can isolate them. It is not possible to calculate the CFS for F1 nor F2, but it is possible to calculate it for F3. Subtracting PB and PC from PA, we eliminate from PA the code from F1, F2, F4, and F5. Eliminating F1 and F2 from PA leaves us with F3. We have found the first isolation operation. Notice that it would be enough to subtract PB from PA to achieve the same result, but we follow the criteria of eliminating the maximum possible CFS expression to get a purer result.

The feature isolation algorithm performs the same search in the rest of the products. In PB, it is possible to isolate its F2 by eliminating F1 and F4 from PC, and it is also possible to isolate its F4 by disposing of F1 and F2 via PA. In PC, we can isolate F5 in a similar fashion as F3 from PA.

At this point, the feature isolation algorithm has gone through all the features of the product family, so the iteration ends. In this iteration, the feature isolation algorithm has calculated the isolation operations for F3(PA), F2(PB), F4(PB), and F5(PC). As there are still features that lack an isolation operation and we have unlocked new isolation operations, the feature isolation algorithm makes a new iteration.

- **Second Iteration:** For all the features in PA that lack an isolation operation, the feature isolation algorithm searches for the CFS that can isolate them. In order to isolate F1, we need to eliminate both F2 and F3. In the first iteration, our algorithm located F2(PB) and F3(PA). They conform the CFS for F1(PA). We can isolate F2(PA) by subtracting PC and F3(PA).

  We can repeat the same steps in both PB and PC. By combining the different products and the features that we isolated in the first iteration, it is possible to get all the isolation operations for the features that lacked them in the previous step (F1(PB), F1(PC), F4(PC)).

  The second iteration has calculated the isolation operations for F1(PA), F2(PA), F1(PB), F1(PC), and F4(PC). At the end of the second iteration, the feature isolation algorithm has isolated all the features, so no more iterations are needed.

As the output of the Stage 2 of the Linked List Example, three tables are returned. Each one of these tables contains the product name, the features that belong to it, and the isolation operations found by the feature isolation algorithm.

3.2.3 Code Comparison

The third stage runs the code comparisons specified by the operations in order to isolate the features in the source code of the products. In a family of software products, the newest products are implemented by carrying out increments or decrements of the previous products in the family. Version control software has become really popular, and there is a wide amount of tool support that calculates differences between two source codes available. Apart from this, code comparison techniques have been used successfully for large scale systems (Kamiya, Kusumoto, and Inoue 2002) (Li et al. 2006), proving the computational cost of the operation to be affordable should we scale up our approach. For all these reasons, we use textual code comparison techniques (diff) to execute the code comparisons dictated by the operations given by the second step of our approach.

The Linked List Example (see 3 Code Comparison of Fig. 4.4) shows how features are isolated. In our approach, all the features isolated at the model level in the second stage are isolated at the code level in the third stage. Due to space restrictions, this example isolates only two features: F2(PB), and F2(PA). According to the operations, F2(PB) can be automatically isolated by subtracting the code belonging to PC from PB. In this example, subtracting the code results in eliminating from PB the inner class Node and the variable declaration section (PB, lines 1 to 8). Therefore, the approach isolates the Sorting Feature from PB (PB, lines 9 to 30).

In order to isolate F2(PA), we must first isolate F3(PA). We subtract both PB and PC to PA, and after eliminating the corresponding code, the approach isolates the Printing Feature (PA, declaration at line 6). We can now isolate F2(PA) by removing from PA the code that is common between PA and PC, and disposing of the F3(PA) code that we just isolated. By doing this, the approach isolates the Sorting Feature from PA (PA, lines 8 to 22). The third stage concludes when the features are isolated in code. The output of the third stage is, for each FN(PX), the code that isolates the feature.

3.2.4 Similarity Comparison

In this stage, the isolated pieces of code that implement the features that belong to more than one product are compared one to one in order to calculate the similarity between them. In order to calculate the similarity between the same feature in two different products, our approach performs a diff between them.

Diff returns the equal parts and the differences in the code of the two features. We discard the code differences and retain the parts of the code that are equal between them. Similarity between features is then measured in terms of the Total Number of Statements (TNOS) (Dit et al. 2013), which is a size metric for measuring code size. TNOS counts the number of statements (e.g. for, if, return, switch, while) in each method for assessing the entire code size. This size metric is not dependent on the coding style of programmers, unlike the Lines Of Code metric.

The Linked List Example (see 4 Similarity Comparison of Fig. 4.4) compares F2(PB) and F2(PA). From the lines of code present in the figure, it can be appreciated that the two order methods, while very similar, do not have the exact same code (notice the marked changes from line 20 to line 28 on PB). It is reasonable, as PA implements a singly linked list and PB implements a doubly linked list. Even if the sorting technique is the same (bubble sort), it cannot be implemented the same way with a different number of links between elements. In fact, F2(PA) has 6 statements and F2(PB) has 7 statements. Considering that 4 of the 7 statements are equal and represent the same conditions in the code, the similarity percentage between F2(PA) and F2(PB) is around the 57%. From this example, we can conclude that some sort of modification has occurred to the feature since it was first implemented on PA until its appearance on PB.

Summarizing, our approach is applied to a family of software products where variability is not formalized. The first stage identifies the features from the products at a model level, tagging the products with them. Then, in the second stage, the operations to isolate the features are calculated. After that, in the third stage, the approach executes the code comparisons dictated by the operations. Finally, in the fourth stage,

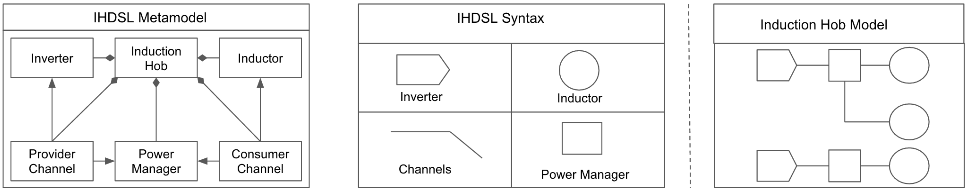


Figure 3.3: IHDSL Metamodel, Syntax and Model

the approach quantifies the degree of similarity between the features that appear in more than one product. Our approach returns, for the different features in the family, the feature isolation at the code level and the degree of similarity between the features that appear in more than one product.

3.3 Evaluation

We have evaluated the presented ideas with our industrial partner (BSH group). Their induction division has been producing induction hobs (under the brands Bosch and Siemens among others) over the last 15 years.

3.3.1 The Induction Hobs Domain

The newest Induction Hobs (IHs) include 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. In addition, there has been an increase in the type of feedback provided to the user while cooking, such as the exact temperature of the cookware, the temperature of the food being cooked, or even real-time measurements of the actual consumption of the IH. All of these changes are being possible at the cost of increasing the software complexity.

The Domain Specific Language used by our industrial partner to specify the Induction Hobs (IHDSL) is composed of 46 meta-classes, 74 references among them and more than 180 properties. However, in order to gain legibility and due to intellectual property rights concerns, in this paper we use a simplified subset of the IHDSL (see Fig. 3.3). The main concepts of IHDSL are: Inverter, Induction Hob, Inductor, Provider Channel, Power Manager and Consumer Channel. The firmware code of each IH is implemented in ANSI C and includes about four hundred thousand TNOS.

In order to gain legibility and due to intellectual property rights concerns, in the following lines, we explain a subset of IHDSL to present the IH domain, although in the evaluation, the complete models have been used. The main concepts of IHDSL are: Inverter, Induction Hob, Inductor, Provider Channel, Power Manager and Consumer Channel.

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 run-time. The path followed by the energy through the channels is controlled by the power manager.

Inductors are the elements where the energy is transformed into an electromagnetic field. Inductors 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 while 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 form at run-time (depending on the cookware placed on top).

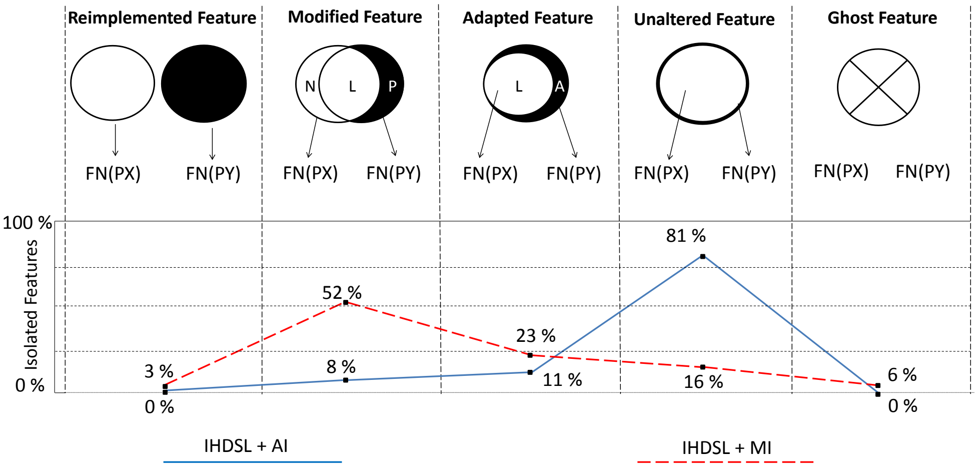


Figure 3.4: Clone-and-own Relationships Extraction applied to both family of products

3.3.2 Extracted Clone-and-Own Relationships

We have applied our Clone-and-Own approach to two families of products of our industrial partner. The first family of products was specified using IHDSL. After the specification, the IH's firmware was manually implemented (MI) in ANSI C by software engineers. This family of products contains a total of 46 products. Since this family of products uses IHDSL and manual implementation we refer to this family as IHDSL+MI. The second family of products was also specified using IHDSL. After the specification, the IH's firmware was automatically implemented (AI) using m2t (model-to-text) transformation. This transformation was produced by Acceleo (Corredor et al. 2012). This family is composed by a total of 66 products. Since this family of products uses IHDSL and automatic implementation we refer to this family as IHDSL+AI.

The IHDSL+MI family has a total of 81 different features. On the other side, the IHDSL+AI family contains a total of 47 features. After applying our Clone-and-Own Relationship extraction approach to both families of products we were able to isolate a total of 49 features belonging to IHDSL+MI and a total of 34 features belonging to IHDSL+AI. As a result, we detected five types of Clone-and-Own Relationships. Given the extracted code of FN(PX) and FN(PY), being product (PX) previous in time to product (PY), and being the same feature (FN) present in both products, we have identified the following feature relationships (see top part of Fig. 3.4).

- **Reimplemented Feature**, FN(PX) and FN(PY) do not share code between them. The implementations of these features are entirely different.
- **Modified Feature**, it exists shared code between both features. The part of code from FN(PX) which is present in FN(PY) is referred to as Legacy. The differences between FN(PX) and the Legacy are referred to as Negative modifications. The differences between FN(PY) and the Legacy are referred to as Positive modifications.
- **Adapted Feature**, FN(PY) includes all code from FN(PX), and additional code which is not present in FN(PX). The part of FN(PX) is referred to as Legacy. Adapter represents the difference between FN(PY) and the Legacy.
- **Unaltered Feature**, the code of FN(PX) and FN(PY) is strictly the same.
- **Ghost Feature**, FN(PY) is specified at the model level but the extraction approach reveals that the code is missing.

We have the intuition that another type of relationship exists, Non-documented Features. Non-documented Features are those features that are not present at the model level, but they are at the code level. Software engineers reported that sometimes they implemented new code in later stages of the development without updating the corresponding IHDSL models. However, the full set of features of neither software family was completely isolated. The unclassified code may belong to either *Non-isolated Features* or *Non-documented Features*. Therefore, we have not evidence that this feature genuinely exists in IHDSL+MI or IHDSL+AI.

3.3.3 Clone-and-Own Relationships for Automatic Implementation

In the IHDSL+AI family our approach extracted the following relationships: 0% Reimplemented, 8% Modified, 11% Adapted, 81% Unaltered and 0% Ghost. The presence of Modified and Adapted Features reveals that the implementation code of those features was refined (Modified Feature) or extended (Adapted Feature) by hand after the execution of the m2t transformation. Each feature classified as Unaltered Feature exhibits the same implementation code across all the members of the family that implement that particular feature. Unaltered Features suggest that the code of those features was not altered by software engineers after the execution of the m2t transformation.

In IHDSL+AI, the presence of Unaltered Features (81%) surpasses the presence of both Modified and Adapted Features (19%). This indicates that the m2t transformation actually saves implementation time to software engineers. Furthermore, the size of Positive modifications is smaller than the size of the Legacy feature on average (Modified Features) and the size of the Adapter is smaller than the Legacy feature on average (Adapted Features). These evidence contributes to concluding that the m2t transformation requires little human intervention.

We suggest that the Modified Feature and Adapted Feature relationships are useful to analyze whether it is necessary to carry out changes over the model-to-code transformation. If it is determined that it is necessary to update it, then the information provided by the occurrences of these relationships can be used to refine the metamodel and the code transformation rules.

In the IHDSL+AI family, modified features enabled to adjust the transformation rules. Negative parts of modified features reflected eliminated code introduced by obsolete transformation rules, and positive parts of modified features reflected manual code additions. The information provided by analyzing both the negative and positive parts enabled the company to update transformation rules with recurring changes that were predicted to keep occurring in the future.

3.3.4 Clone-and-Own Relationships for Manual Implementation

In the IHDSL+MI family our approach extracted the following relationships: 3% Reimplemented, 52% Modified, 23% Adapted, 16% Unaltered and 6% Ghost. The presence of Modified and Adapted Features reveals that the implementation code was reused from another product as source and then refined to meet the particularities of the target product. F2(PA) and F2(PB) of the Linked List example (see Fig. 4.4) are instances of the Modified Feature relationship. On one hand, both F2(PA) and F2(PB) implement the same functionality (sorting the lists using the bubble method). On the other hand the implementation details of F2(PA) are different than those of F2(PB) to accommodate a feature (F4 = Backwards Linking) of PB which is not a feature of PA.

Unaltered Features were copied from previous products and used directly in new products. It turns out, Unaltered Features are reused among different products without requiring refinements on part of the engineer to accommodate the rest of the features of the product.

In IHDSL+MI, Unaltered, Adapted and Modified Features (91%) reveal reuse opportunities identified by the software engineers. The presence of Reimplemented Features (3%) indicates that software engineers did not realize former implementations of the feature. The implementation of these features was done from scratch, revealing missed reuse opportunities. Finally 6% of isolated features were cataloged as Ghost Features. Ghost Features reveal inconsistencies between the model specification and the implemented code. The model specification should be updated to keep software engineers from failing to locate the code of those features.

We suggest that Reimplemented Feature relationships are useful to detect feature reuse impediments. In IHDSL+MI, for instance, they were useful to detect that a developer had left the company without performing knowledge transfer, and that the new developer in his place eventually reimplemented some code from scratch. Apart from detecting the situation, now we have awareness of both implementations, therefore widening the reuse possibilities.

We propose that Modified Feature and Adapted Feature relationships are useful for analyzing cost-benefit payoffs of reusing code fragments against reimplementing them. In IHDSL+MI, for instance, 12 cases were found where it had become more costly to create adapters that allowed reusing the legacy part of a feature than to reimplement the feature as needed.

We propound that Unaltered Feature relationships are useful to detect the opportunities to improve the reuse maturity of a family of software products. In IHDSL+MI, for instance, they were useful to build an implementation framework that has been used in further developments.

3.3.5 Limitations

There are some limitations that must be acknowledged. To begin with, there are companies that implement the code directly from the software requirements. This leads to software product families implemented without models. In such an scenario, our approach is not applicable. Developing and using techniques that permit to carry out feature location at the requisites level would widen the scope of our approach.

Second, depending on the configuration of the products in the software family, it is possible for our feature isolation algorithm to not find the isolation operations for every feature in every product. In the future, our approach might suggest the addition of products to the family with specific feature configurations that would allow the algorithm to isolate non-isolated features.

In addition, determining the kind of Clone-and-Own Relationships between products entails some degree of uncertainty. Specifically in the cases of reimplementation and feature modification, the current criteria is very rigid. This results in reimplemented features that, due to having low amounts of common code, are incorrectly classified as modified ones.

Finally, inspecting the isolated features with domain experts, we detected that in some cases, not all the lines of code provided in an isolated piece of code belong to the isolated feature and, in some other cases, some lines that do belong to the isolated feature are missing. Nevertheless, we have confirmed that the isolated code is a good heuristic for feature location, and domain experts have validated that the behavior detected by the described Clone-and-Own Relationships is the right one at the code level.

3.4 Related Work

Approaches related to the one presented in this paper can be distinguished into two areas: feature location at the model level and feature location at the code level. First we introduce the state-of-the-art of feature location at the code level and secondly, the state-of-the-art of feature location at the model level.

3.4.1 Feature Location at the Code Level

Some works apply type systems to extract relevant information when constructing the variability model. For instance, Typechef (Kästner, Giarrusso, et al. 2011) provides an infrastructure to analyze the variability with the #ifdef directives. In (Kästner, Ostermann, and Erdweg 2012) the authors extend Typechef in order to support the variability at runtime.

Text similarity techniques are based on mathematical methods to determine the similarity in a collection of texts. As an example, Latent Semantic Indexing (LSI) (Landauer and Psotka 2000) takes into account the number of occurrences in a set of words in large texts. LSI can be used to obtain similarity measurement metrics between features and the code used to implement them. These similarity can be represented by Vector Space Models (VSM). On some occasions text similarity techniques are combined with dynamic analysis (Asadi et al. 2010).

Other works focus on applying reverse engineering to the source code to obtain the variability model (Czarnecki and Wasowski 2007), (She et al. 2011). In (Czarnecki and Wasowski 2007) the authors use propositional logic which describes the dependencies between features. In (Nadi et al. 2014) Typechef and propositional logic are used to extract conditions among a collection of features.

Several approaches (Walkinshaw, Roper, and Wood 2007), (Trifu 2009) apply Program Dependence Analysis (PDA) to locate features. PDA can be represented by Program Dependence Graphs (PDG) where the nodes represent functions or global variables and the edges represent function calls or accesses to global variables.

Trace analysis is a run-time technique used to define a variability model through relevant information. When the technique is executed, it produces traces indicating which parts of code have been executed. Some approaches (Eisenberg and Volder 2005) are based on traces analysis. There are also works that combine dynamic analysis and static analysis as is the case of LSI (Poshyvanyk et al. 2007), PDA (Eisenberg and Volder 2005) or VSM (Eaddy et al. 2008).

Compared to the above works, our approach introduces software models as a new source of knowledge for feature location at the code level. Furthermore, our approach not only isolates the implemented code of the features but it also extracts Clone-and-Own Relationships among these features. These relationships are used to better understand how features are reused, and to suggest improvements on the way they are reused.

3.4.2 Feature Location at the Model Level

In (Rubin and Chechik 2012), the authors propose a framework for mining legacy product lines and automating their refactoring to contemporary feature-oriented SPLE approaches. They compare the elements of the input with each other, matching those whose similarity is above a certain threshold and merging them together. In (Zhang, Haugen, and Møller-Pedersen 2011), the authors propose a generic approach to automatically compare products and extract the variability among them in terms of Common Variability Language (CVL) (Haugen et al. 2008), (Svendsen et al. 2010). In (Font, Arcega, et al. 2015) an approach to automate the formalization of variability in a given family of models is presented. The model commonalities and differences are specified as placements over a base model and replacements in a model library. The resulting Software Product Line (SPL) enables the derivation of new product models by reusing the extracted model fragments. In (Martinez et al. 2015) the authors propose another approach based on comparisons to extract the variability of any kind of asset. These works focus on formalizing the variability in a SPL. Finally, (Font, Ballarin, et al. 2015) identifies model patterns in a set of models and conceptualizes the extracted patterns as reusable model fragments.

The above approaches limit their application to finding fragments of a model which represent features in order to formalize the variability in a SPL. In contrast, our approach combines feature location at the model level with code comparison in order to isolate the implemented code of the features. Furthermore, our work identifies several different Clone-and-Own Relationships among the located features. These relationships enable us to make improvement suggestions based on the knowledge gathered on the way features are reused.

3.5 Conclusions

To keep pace with the increasing demand for custom-tailored software systems, companies often apply the clone-and-own practice, through which a new product in a software product family is built by copying and adapting code from other products in the family.

In this work, we show our approach, which leverages feature location to identify and extract the Clone-and-Own Relationships from a family of software products. We have proposed an approach that extracts the features at the model level and, with that information, calculates isolation operations that enable to isolate the features at the code level. This work allows us to isolate the features of the different products in the code. With the achieved code isolation, features are compared at the code level in order to define the relationships between them.

We have evaluated the approach with our industrial partner, extracting the Clone-and-Own Relationships presented in two product families of induction hob models. One of the families had its code implemented manually and the other one, in an automatic way.

A total of five different relationships have been extracted. These relationships entitle Reimplemented, Modified, Adapted, Unaltered, and Ghost Features. The results of our approach provide insight into understanding the Clone-and-Own relationships of the features in a family of software products. These relationships are then used to suggest improvements on how features are reused.

In the case of families where automatic code generation is applied, the Modified and Adapted Features are used to analyze whether it is necessary to carry out changes over the model-to-code transformation. If it is determined that it is necessary to improve it, then the information provided by the occurrences of these relationships can be used to refine the metamodel and the code transformation rules.

In the case of families where the code is manually implemented, Reimplemented Features are used to detect feature reuse impediments; Modified and Adapted Features are used for analyzing cost-benefit payoffs of reusing code fragments against reimplementing them; and Unaltered Features are used to detect opportunities to improve the reuse maturity of a family of software products.

Bibliography

Asadi, Fatemeh et al. (2010). “A Heuristic-Based Approach to Identify Concepts in Execution Traces”. In: *14th European Conference on Software Maintenance and Reengineering, CSMR, March ’10, Madrid, Spain.* Ed. by Rafael Capilla, Rudolf Ferenc, and Juan C Dueñas. IEEE Computer Society. ISBN: 978-0-7695-4321-5. DOI: 10.1109/CSMR.2010.17 (cit. on p. 99).

Corredor, Iván et al. (2012). “Model-Driven Methodology for Rapid Deployment of Smart Spaces Based on Resource-Oriented Architectures”. In: *Sensors.* ISSN: 1424-8220. DOI: 10.3390/s120709286 (cit. on p. 94).

Czarnecki, Krzysztof and Andrzej Wasowski (2007). “Feature Diagrams and Logics: There and Back Again”. In: *Software Product Lines, 11th International Conference, SPLC, Kyoto, Japan, September 10-14, 2007, Proceed-*

ings. IEEE Computer Society. DOI: 10.1109/SPLINE.2007.24 (cit. on pp. 83, 99).

Dit, Bogdan et al. (Jan. 2013). "Feature location in source code: A taxonomy and survey". In: *Journal of Software Maintenance and Evolution: Research and Practice* 25. DOI: 10.1002/smr.567 (cit. on pp. 83, 85, 91).

Eaddy, Marc et al. (2008). "CERBERUS: Tracing Requirements to Source Code Using Information Retrieval, Dynamic Analysis, and Program Analysis". In: *The 16th IEEE International Conference on Program Comprehension, ICPC, Amsterdam, The Netherlands, June 10-13, 2008*. Ed. by René L Krikhaar, Ralf Lämmel, and Chris Verhoef. IEEE Computer Society, pp. 53–62. DOI: 10.1109/ICPC.2008.39 (cit. on pp. 83, 100).

Eisenberg, Andrew David and Kris De Volder (2005). "Dynamic Feature Traces: Finding Features in Unfamiliar Code". In: *21st IEEE International Conference on Software Maintenance (ICSM), 25-30 September 2005, Budapest, Hungary*. IEEE Computer Society, pp. 337–346. ISBN: 0-7695-2368-4. DOI: 10.1109/ICSM.2005.42 (cit. on p. 100).

Font, Jaime, Lorena Arcega, et al. (2015). "Building Software Product Lines from Conceptualized Model Patterns". In: *Proceedings of the 19th International Conference on Software Product Line*. SPLC '15. New York, NY, USA: ACM. ISBN: 978-1-4503-3613-0 (cit. on pp. 85, 87, 100).

Font, Jaime, Manuel Ballarin, et al. (July 2015). "Automating the variability formalization of a model family by means of common variability language". In: pp. 411–418. DOI: 10.1145/2791060.2793678 (cit. on pp. 83, 101).

Haugen, Øystein et al. (2008). "Adding Standardized Variability to Domain Specific Languages". In: *Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, 2008, Proceedings*. IEEE Computer Society, pp. 139–148. ISBN: 978-0-7695-3303-2. DOI: 10.1109/SPLC.2008.25 (cit. on p. 100).

Kamiya, Toshihiro, Shinji Kusumoto, and Katsuro Inoue (2002). "CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code". In: *IEEE Trans.* DOI: 10.1109/TSE.2002.1019480 (cit. on p. 90).

Kästner, Christian, Paolo G Giarrusso, et al. (2011). "Variability-aware parsing in the presence of lexical macros and conditional compilation". In: *Proceedings of the 26th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2011.* Ed. by Cristina Videira Lopes and Kathleen Fisher. ACM. ISBN: 978-1-4503-0940-0 (cit. on p. 99).

Kästner, Christian, Klaus Ostermann, and Sebastian Erdweg (2012). "A variability-aware module system". In: *Proceedings of the 27th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, USA, October 21-25, 2012.* Ed. by Gary T Leavens and Matthew B Dwyer. ACM. ISBN: 978-1-4503-1561-6 (cit. on p. 99).

Landauer, Thomas K and Joseph Psotka (2000). "Simulating Text Understanding for Educational Applications with Latent Semantic Analysis: Introduction to LSA". In: *Interactive Learning Environments.* DOI: 10.1076/1049-4820(200008)8:2;1-B;FT073 (cit. on p. 99).

Li, Zhenmin et al. (2006). "CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code". In: *IEEE Trans. Software Eng.* DOI: 10.1109/TSE.2006.28 (cit. on p. 90).

Martinez, Jabier et al. (2015). "Bottom-up adoption of software product lines: a generic and extensible approach". In: *Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015.* Ed. by Douglas C Schmidt. ACM. ISBN: 978-1-4503-3613-0 (cit. on pp. 83, 85, 87, 100).

Nadi, Sarah et al. (2014). "Mining configuration constraints: static analyses and empirical results". In: *36th International Conference on Software Engineering, ICSE 14, Hyderabad, India - May 31 - June 07, 2014.* Ed. by Pankaj Jalote, Lionel C Briand, and André van der Hoek. ACM, pp. 140–151. ISBN: 978-1-4503-2756-5. DOI: 10.1145/2568225.2568283 (cit. on p. 99).

Poshyvanyk, Denys et al. (2007). "Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval". In: *IEEE Trans. Software Eng.* DOI: 10.1109/TSE.2007.1016 (cit. on p. 100).

Rubin, Julia and Marsha Chechik (2012). “Combining Related Products into Product Lines”. In: *Fundamental Approaches to Software Engineering - 15th International Conference, FASE 2012 Tallinn, Estonia, March 24 - April 1, 2012. Proceedings.* Ed. by Juan de Lara and Andrea Zisman. Vol. 7212. Lecture Notes in Computer Science. Springer, pp. 285–300. ISBN: 978-3-642-28871-5 (cit. on pp. 83, 85, 87, 100).

Selic, Bran (2003). “The Pragmatics of Model-Driven Development”. In: *IEEE Software.* DOI: `10.1109/MS.2003.1231146` (cit. on p. 84).

She, Steven et al. (2011). “Reverse engineering feature models”. In: *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011.* Ed. by Richard N Taylor, Harald C Gall, and Nenad Medvidovic. ACM. ISBN: 978-1-4503-0445-0. DOI: `10.1145/1985793.1985856` (cit. on p. 99).

Svendsen, Andreas et al. (2010). “Developing a Software Product Line for Train Control: A Case Study of CVL”. In: *Software Product Lines: Going Beyond - 14th International Conference, SPLC, Jeju Island, South Korea, September 13-17, 2010. Proceedings.* Ed. by Jan Bosch and Jaejoon Lee. Vol. 6287. Lecture Notes in Computer Science. Springer, pp. 106–120. ISBN: 978-3-642-15578-9 (cit. on p. 100).

Trifu, Mircea (2009). “Improving the Dataflow-Based Concern Identification Approach”. In: *13th European Conference on Software Maintenance and Reengineering, CSMR 2009, Architecture-Centric Maintenance of Large-SCale Software Systems, Kaiserslautern, Germany, 24-27 March 2009.* Ed. by Andreas Winter, Rudolf Ferenc, and Jens Knodel. IEEE Computer Society. ISBN: 978-0-7695-3589-0. DOI: `10.1109/CSMR.2009.34` (cit. on p. 100).

Walkinshaw, Neil, Marc Roper, and Murray Wood (2007). “Feature Location and Extraction using Landmarks and Barriers”. In: *23rd IEEE International Conference on Software Maintenance (ICSM 2007), October 2-5, 2007, Paris, France.* IEEE. ISBN: 978-1-4244-1256-3. DOI: `10.1109/ICSM.2007.4362618` (cit. on p. 100).

Zhang, Xiaorui, Øystein Haugen, and Birger Møller-Pedersen (2011). “Model Comparison to Synthesize a Model-Driven Software Product Line”. In:

Software Product Lines - 15th International Conference, SPLC, Munich, Germany, August 22-26, 2011. Ed. by Eduardo Santana de Almeida et al. IEEE, pp. 90–99. ISBN: 978-1-4577-1029-2 (cit. on pp. 85, 87, 100).

Chapter 4

Locating Clone-and-Own Relationships in Model-based Industrial Families of Software Products to Encourage Reuse

Companies often develop similar product variants that share a high degree of functionality (i.e., features) by copying and modifying code (the clone-and-own approach). In an industrial context with a large amount of variants, software reuse can become complex for engineers. Identifying the clone-and-own relationships between the same feature in different product variants can encourage reuse (e.g., suggesting improvements on how features are reused or detecting feature reuse impediments). This work presents our approach to locate the clone-and-own relationships. To do this, our approach proposes an algorithm that combines feature location and code-comparison techniques. We evaluated our approach in three model-based industrial families of two domains (firmware for induction hobs and train control software).

In our evaluation, we measure the performance (in terms of precision and recall) and we compare our approach with its previous version (baseline), which uses a different technique to compare the code of each feature with its variants. The results show that our approach is able to locate clone-and-own relationships in different domains of real world environments, and it outperforms the baseline up to 65.37% in terms of precision.

4.1 Introduction

Recent research has pointed out that a family of software products inevitably contains a large amount of similar code (Pham et al. 2012) that could be reused. Companies often develop a portfolio of similar product variants which share a high degree of common functionality (i.e., features) and code, mostly due to the copy-and-paste programming practice (the *clone-and-own* approach). In an industrial context, engineers could face thousands of products that share features among them, so software maintenance and reuse could be complex.

Identifying the clone-and-own relationships across the family of products can encourage reuse. For example, clone-and-own relationships can help developers to suggest improvements on how features are reused, detect feature reuse impediments, analyze the cost-benefit payoffs of reusing code fragments against reimplementing them, and detect the maturity of a family of software products.

To encourage reuse, previous approaches have been proposed to locate features from the source code or the models of a family of products. At the code level, there are approaches (Landauer and Psotka 2000; Asadi et al. 2010; Czarnecki and Wasowski 2007; She et al. 2011) that isolate the implementation of the features but they do not extract the clone-and-own relationships among features. In (Fischer et al. 2014), associations between artifacts are obtained by comparing the source code of existing product variants to provide hints at what features could not be separated, or for which artifacts there are multiple order options available. In (Lin et al. 2017), templates are extracted from recurring designs in source code. However, these approaches target code and do not leverage models as a source of feature location knowledge. Models have been proved to increase efficiency and effectiveness in software development (Brambilla, Cabot, and Wimmer 2012). Therefore, companies that develop their software products using models cannot apply these approaches to encourage reuse. At the model level, approaches target the formalization of the variability in the family of products to encourage the reuse of model fragments (Rubin and Chechik 2012; Zhang, Haugen, and Møller-Pedersen 2011; Font, Arcega, et al. 2015; Martinez et al. 2015). However, these approaches do not incorporate both feature location at model level and comparisons at code level with the goal of isolating implementations of individual features.

To cope with this lack, we propose an approach that locates the clone-and-own relationships between features in a model-based family of software products, reflecting how features are reused throughout its development. Our approach first leverages the information that existing techniques on feature location provide in order to develop an algorithm that is able to retrieve the code associated with each feature. Afterwards, our approach compares the source code of an isolated feature in a particular product against the source code of the different isolations of the same feature in other products, locating the clone-and-own relationships between the different feature isolations.

To show the feasibility and generalization of our approach, we have applied it in three industrial model-based families of software products from two domains: two model-based families of firmware for induction hobs provided by our industrial partner BSH, and a model-based family of train control PLC software provided by our industrial partner CAF. The BSH[1] group produces firmwares for their induction hobs (sold under the brands of Bosch and Siemens) over more than 15 years. CAF[2] produces PLC software to control the trains that they manufacture over more than 25 years.

The results of our evaluation show that our approach can be applied in different domains of real world environments and it is able to locate the following clone-and-own relationships between features: Reimplemented, Modified, Adapted, Unaltered, and Ghost Features. In addition, the results show that our approach outperforms the baseline in terms of precision for modified (up to 65.37%) and adapted (up to 37.5%) clone-and-own relationships, and in terms of recall for adapted (up to 48.72%) and unaltered (up to 17.95%) relationships thanks to the improved code comparison between features, which avoids that unaltered clone-and-own relationships are incorrectly classified as adapted or modified, and adapted clone-and-own relationships are incorrectly classified as modified.

[1] www.bsh-group.com

[2] www.caf.net/en

This paper is an extension of a conference paper (Ballarin, Lapeña Martí, and Cetina 2016) and the significant differences with the conference version include: 1) The modification of the step of our approach that compares the source code of a feature in a product with the source code of the same feature in another product in order to avoid irrelevant textual differences; 2) The application of our approach in a different industrial domain (the train control PLC software provided by CAF) in order to prove its generalization; and 3) The evaluation has been further extended to measure the performance of both our approach and the baseline (the previous version of our approach) in terms of recall and precision in the three industrial case studies.

The remainder of the paper is structured as follows: Section 4.2 provides a background of the clone-and-own relationships. Section 4.3 presents our approach and shows how to apply it to a simple example. Section 4.4 shows the evaluation of our approach in three case studies of two industrial domains. Section 4.5 summarizes the related work, and Section 4.6 states the relevant conclusions.

4.2 Background

This section presents the different clone-and-own relationships as well as how these relationships can help developers to suggest improvements on how features are reused.

Figure 4.1 shows an example of a family of software products. Product A consists of two features (F1 and F2). After some time, another product (Product B) is constructed from a variant of F1 from Product A (using the clone-and-own approach), so Product B holds a clone-and-own (CAO) relationship with a previous product, Product A. Moreover, Product B comprehends a new feature (F3), which has been created from scratch. After, another product (Product C) is built with a new feature (F4), a variant of F3 from Product B, and a variant of F2 from Product A. Hence, Product C holds two clone-and-own relationships with Product A and Product B, one for each reused feature. In total, this family of products comprises 3 products, 4 features, and 3 clone-and-own relationships.

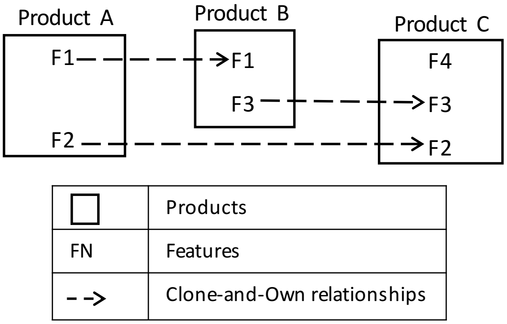


Figure 4.1: Clone-and-own relationships in a family of products

Different clone-and-own relationships may exist in a product family. The existing relationships depend on the reuse possibilities of FN(PX) and FN(PY), where (PX) is a product that existed priorly to another product (PY), and where (FN) is a feature that is present in both (PX) and (PY) (e.g., F1(PA) and F1(PB) in Figure 4.1). We identified in (Ballarin, Lapeña Martí, and Cetina 2016) the clone-and-own relationships that Figure 4.2 depicts:

1. **Reimplemented Feature:** There is no shared code between FN(PX) and FN(PY). Therefore, their implementations are entirely different.

2. **Modified Feature:** There is, to some extent, code that is shared between both features. Code from FN(PX) that is also present in FN(PY) is denoted as Legacy. Differences among FN(PX) and FN(PY) are denoted as modifications.

3. **Adapted Feature:** FN(PY) includes all the code from FN(PX), plus additional novel code. Code of FN(PX) is denoted as Legacy. The novel code that causes FN(PY) and the Legacy to differ is denoted as Adapter.

4. **Unaltered Feature:** The implementations of FN(PX) and FN(PY) contain the exact same code.

5. **Ghost Feature:** The FN feature is theoretically included in PY, but the approach uncovers that the code of FN is not present in PY.

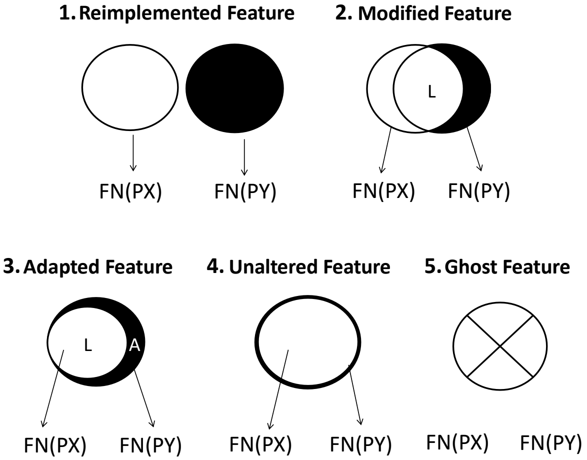


Figure 4.2: Types of clone-and-own relationships

The identified clone-and-own relationships may assist developers suggest improvements on feature reuse in the following ways: (1) **Reimplemented Feature** relationships help detect feature reuse barriers, indicating the existence of former implementations of features that were unrecognized by software engineers and therefore recreated from scratch, revealing missed reuse opportunities; (2) **Modified Feature** and (3) **Adapted Feature** relationships aid on the analysis of the cost-benefit trade-offs of code fragment reuse opposite to code fragment reimplementation; (4) **Unaltered Feature** relationships help detect chances to improve the reuse maturity of a software product family; and (5) **Ghost Feature** relationships highlight discrepancies between the requirements and the implementations, and therefore, the specification should be amended to refrain software engineers from wasting time trying to locate the code of those features for reuse.

For instance, Reimplemented Feature relationships may denote that a software engineer terminated his contract without transferring his knowledge of the software (Ballarin, Lapeña Martí, and Cetina 2016), eventually causing a fresh development of an already existing feature by another software engineer in his place. In addition to the discovery of the situation, the relationship raises awareness on both implementations, broaden-

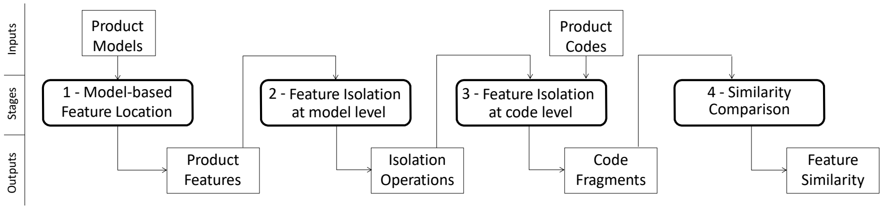

Inputs
Stages
Outputs
Product Models
1 - Model-based Feature Location
Product Features
2 - Feature Isolation at model level
Isolation Operations
Product Codes
3 - Feature Isolation at code level
Code Fragments
4 - Similarity Comparison
Feature Similarity

Figure 4.3: Overview of our approach

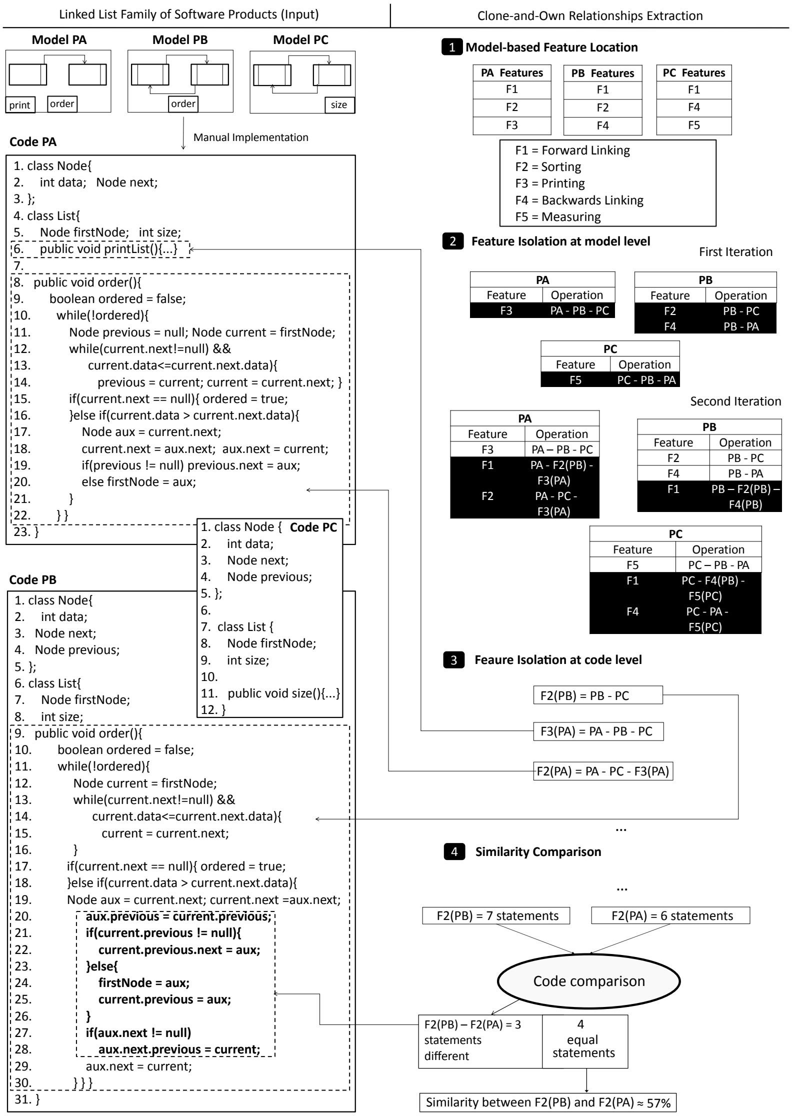
ing the reuse possibilities. Furthermore, Unaltered Feature relationships can be utilized to assemble an implementation framework that can help in the construction of future developments.

4.3 The Clone-and-Own Extraction Approach

Our approach takes as input the product models that specify a family of software products, and the product codes obtained as a result of either the translation of the models by developers or the automatic translation using a model-to-text transformation (Selic 2003). Next, our approach extracts Clone-and-Own Relationships in order to enable developers to understand and improve how features are reused among the products. Our approach builds up on feature location at the model level and code comparisons.

Figure 4.3 depicts the four main stages of our approach (Model-based Feature Location, Feature Isolation at model level, Feature Isolation at code level and Similarity Comparison) as well as the inputs and outputs of these stages.

We exemplify our work through the Linked List running example, based on a software products family where the variability is undefined. The products of the family have linked models, in which the code has been manually developed by a human (see left side of Figure 4.4). The products are either singly or doubly linked lists. Each one has a different mixture of added functionality: functionality that prints the elements of the list, functionality to sort the list through the bubble algorithm, and functionality to calculate the amount of elements of the list.


Linked List Family of Software Products (Input)
Clone-and-Own Relationships Extraction
Model PA
Model PB
Model PC
print
order
order
size
Manual Implementation
1 Model-based Feature Location
2 Feature Isolation at model level
First Iteration
Second Iteration
3 Feaure Isolation at code level
4 Similarity Comparison
...
...
Code comparison

Code PA

```
class Node{
    int data;   Node next;
};
class List{
    Node firstNode;   int size;
    public void printList(){...}

public void order(){
    boolean ordered = false;
      while(!ordered){
          Node previous = null; Node current = firstNode;
          while(current.next!=null) &&
                current.data<=current.next.data){
                  previous = current; current = current.next; }
          if(current.next == null){ ordered = true;
          }else if(current.data > current.next.data){
              Node aux = current.next;
              current.next = aux.next;  aux.next = current;
              if(previous != null) previous.next = aux;
              else firstNode = aux;
          }
      } }
}
```

Code PC

```
class Node {
    int data;
    Node next;
    Node previous;
};

class List {
    Node firstNode;
    int size;

    public void size(){...}
}
```

Code PB

```
class Node{
   int data;
  Node next;
  Node previous;
};
class List{
   Node firstNode;
   int size;
public void order(){
      boolean ordered = false;
      while(!ordered){
          Node current = firstNode;
          while(current.next!=null) &&
                current.data<=current.next.data){
                  current = current.next;
          }
        if(current.next == null){ ordered = true;
        }else if(current.data > current.next.data){
        Node aux = current.next; current.next =aux.next;
            aux.previous = current.previous;
            if(current.previous != null){
                current.previous.next = aux;
            }else{
                firstNode = aux;
                current.previous = aux;
            }
            if(aux.next != null)
                aux.next.previous = current;
            aux.next = current;
        }}}
}
```

PA Features
F1
F2
F3

PB Features
F1
F2
F4

PC Features
F1
F4
F5

F1 = Forward Linking
F2 = Sorting
F3 = Printing
F4 = Backwards Linking
F5 = Measuring

First Iteration

PA	
Feature	Operation
F3	PA - PB - PC

PB	
Feature	Operation
F2	PB - PC
F4	PB - PA

PC	
Feature	Operation
F5	PC - PB - PA

Second Iteration

PA	
Feature	Operation
F3	PA – PB - PC
F1	PA - F2(PB) - F3(PA)
F2	PA - PC - F3(PA)

PB	
Feature	Operation
F2	PB - PC
F4	PB - PA
F1	PB – F2(PB) – F4(PB)

PC	
Feature	Operation
F5	PC – PB - PA
F1	PC - F4(PB) - F5(PC)
F4	PC - PA - F5(PC)

F2(PB) = PB - PC
F3(PA) = PA - PB - PC
F2(PA) = PA - PC - F3(PA)

F2(PB) = 7 statements
F2(PA) = 6 statements

F2(PB) – F2(PA) = 3 statements different
4 equal statements

Similarity between F2(PB) and F2(PA) ≈ 57%

Figure 4.4: The Linked List example to show the extraction of Clone-and-own relationships

Each stage of our approach is described in the following subsections.

4.3.1 Model-based Feature Location

In the first stage of our approach, features are extracted from the models of the products. Given a set of models, already existing Feature Location techniques can be leveraged to identify features in the models. Feature Location (identifying a fragment of source code or software model, corresponding to a specific functionality) is one of the most frequent maintenance activities undertaken by developers (Dit et al. 2013).

Several research works in literature tackle feature location in models (Zhang, Haugen, and Møller-Pedersen 2011; Martinez et al. 2015; Rubin and Chechik 2012). For our work, we adopted Conceptualized Model Patterns to Feature Location (CMP-FL) (Font, Arcega, et al. 2015). CMP-FL identifies model patterns by human-in-the-loop (that is, through the domain knowledge of experts and engineers who participate in the process) and then conceptualizes the extracted patterns as reusable model fragments. We adopted this technique since it allows humans to be involved in the extraction process, which improves the results since it makes that the model fragments obtained are more recognizable for humans than the model fragments obtained through automatic approaches (Font, Arcega, et al. 2015).

Through CMP-FL, the elements that differ between the models are considered as alternatives for a feature, and the elements from a model that do not have a match in the rest of the models are extracted as optional features. As a result, the models are broken down into reusable fragments. Each of these reusable fragments will correspond with one of the features of the software products family. The output of the first stage of our approach is a collection of the features located in the models that belong to each product.

For example, the Linked List example of Figure 4.4 (see 1 in the upper-right part of the figure) tags the products (PA, PB, and PC) with the names associated with the located features (F1-F5). In the example, five features are identified within the product family. In product PA, features

F1, F2, and F3 are detected. In product PB, features F1, F2, and F4 are detected. Finally, in product PC, features F1, F4, and F5 are detected.

Current techniques that locate features at the model level (Zhang, Haugen, and Møller-Pedersen 2011), (Martinez et al. 2015), (Rubin and Chechik 2012) (Font, Arcega, et al. 2015) do not provide meaningful names, only synthetic names (such as F1 or F2, for instance). We have decided to add more meaningful names to the features in order to improve the understanding of the example as the figure shows: F1 represents the Forward Linking feature, F2 represents the Sorting feature, F3 represents the Printing feature, F4 represents the Backwards Linking feature, and F5 represents the Measuring feature.

Notice that some of the features are present in more than one product: for instance, feature F2 is present in both the PA and PB products. To avoid ambiguity in the names of the features, a feature FN that belongs to a product PX will be referred to as FN(PX). For example, F2(PA) refers to F2 of PA.

4.3.2 Feature Isolation at model level

This second stage takes as input the list of the existing products and their features (which has been obtained in the previous stage) in order to perform subtractions between the different products at the model level, with the aim of isolating the code of individual features. To that extent, we developed an algorithm that determines the features that can be isolated at the model level. Each feature is accompanied by one operation, which expresses the code subtractions that need to be carried out between products to isolate the feature. The implementation of the algorithm is described through the following paragraphs.

- First of all, the algorithm creates an empty list, used to save the features that can be isolated.
- Then, the algorithm calculates the Complementary Feature Set (CFS) for each feature FN of every product PX. The CFS is a product, combination of products, or combination of products plus already isolated features, that contains all the features in PX except for FN.

A CFS that contains features that are not present in PX is still valid. Subtracting the calculated CFS to PX results in the isolation of FN. With the definition of the CFS, the isolation operation is built as $FN(PX) = PX - CFS$.

- The isolated features are added to the list of isolated features along with their isolation operations. The addition of new features to the list enables for new CFS, and hence, for new feature isolation possibilities. Therefore, the algorithm performs iterations while new features are added to the isolated features list.

In the first iteration, the features that can be isolated by a CFS built through a single product or through a combination of products are included into the list. The operations found via the first iteration establish the base cases of the algorithm. In the iterations that follow, combinations between products and already calculated features serve as the CFS. The isolation operations that are found in this manner form the recursive cases of the algorithm.

Following the Linked List Example, the right side of Figure 4.4, part 2, shows the application of two iterations of the described algorithm:

- **First Iteration:** For all the features in PA (that is, features F1, F2, and F3), the algorithm searches for their CFS, which is only possible to calculate for F3: by removing PB and PC from PA, the code from F1, F2, F4, and F5 is eliminated from PA. Removing F1 and F2 from PA leaves us with F3, and thus, the first isolation operation is found. Notice that, while it would be enough to subtract PB from PA to achieve the same result, the criteria of eliminating the maximum possible CFS expression is followed, in order to get a purer result.

  The algorithm performs the same operation in all the products. In PB, it is possible to isolate F2 by eliminating F1 and F4 from PC, and it is also possible to isolate F4 by disposing of F1 and F2 through the removal of PA. In PC, we can isolate F5 in the same way as F3 is isolated from PA. At this point, the algorithm has gone through all the features of the family of software products, ending the iteration.

As a result of the first iteration, the algorithm has finally retrieved the operations for F3(PA), F2(PB), F4(PB), and F5(PC). Since there are features that still lack an isolation operation, and since new isolation operations have been discovered in the iteration, the algorithm performs a new iteration.

- **Second Iteration:** The algorithm searches for the CFS that can isolate all the features in PA that lack an isolation operation. In order to isolate F1(PA), the algorithm removes F2 and F3 through F2(PB) and F3(PA). Moreover, F2(PA) can be isolated by subtracting PC and F3(PA) from PA.

  The same steps are followed in PB and PC. Through combinations of the different products and the features that were previously isolated, it is possible to get the isolation operations for the features that have not been isolated yet (F1(PB), F1(PC), F4(PC)).

  Therefore, the second iteration of the algorithm has produced the isolation operations for features F1(PA), F2(PA), F1(PB), F1(PC), and F4(PC). Therefore, the algorithm has isolated all the features at this point, rendering a third iteration unnecessary.

At the end, three tables are returned as output of Stage 2 (Feature Isolation at model level) of the Linked List Example (see 2 of the right part of Figure 4.4). Each table contains: the product name, the features that belong to it, and the isolation operations found by the algorithm.

4.3.3 Feature Isolation at code level

This third stage performs the feature isolation at the code level so as to isolate the source code of the features in the products. In a software products family, novel products are implemented through increments or decrements of already existing family products. Version control software is really popular nowadays, and a wide amount of tool support for calculating differences between two source codes is available. Moreover, code comparison techniques have been used with success for large scale systems (Kamiya, Kusumoto, and Inoue 2002; Li et al. 2006), being the computational cost of the operation affordable should we scale up our approach. Due to all these reasons, we use diff techniques (textual comparisons) to perform code comparisons in this stage.

Following the Linked List example, features that were isolated at the model level in the second stage are now isolated at the code level. Right side of Figure 4.4, part 3, shows as an example the isolation of features F2(PB), F3(PA) and F2(PA). According to operation F2(PB) = PB-PC, F2(PB) can be isolated through subtracting the code of PC from PB (lines 1 to 8). Hence, the approach isolates F2 from PB (lines 9 to 30).

In order to isolate F2(PA), it is necessary to isolate F3(PA) first according to the operation F2(PA)=PA-PC-F3(PA). To do this, PB and PC are subtracted from PA according to the operation F3(PA)= PA-PB-PC. The result is the isolation of the F3 (the Printing feature from PA, declared at line 6). After, F2(PA) can be isolated by removing the code that is common between PA and PC from PA, thus removing the F3(PA) code that we just isolated. As a result, the approach isolates the F2 from PA (Sorting feature, lines 8 to 22). This stage comes to an end when the code of the features is isolated. The final output of the algorithm is the retrieved group of code fragments (one for each isolated feature).

4.3.4 Similarity Comparison

In this last stage, the isolated code fragments that implement the features, which are in more than one product, are compared one to one in order to calculate the similarity between them. To do this, our approach performs a comparison between two fragments of code that implement the same feature.

To avoid the detection of some irrelevant textual differences in our approach, we use the technique described in (Horwitz 1990) that computes semantic and textual differences between two programs. Although this technique does not determine precisely the set of semantic changes since it is currently limited to scalar variables, assignment statements, conditional statements, while loops, and output statements, it could detect renaming local variables as textual differences and it does not flag different extra spaces and line breaks as differences.

This technique first tries to match every component of a *New* version of a code fragment with an *Old* version that is both semantically and textually equivalent. Next, the procedure considers all unmatched components of New, attempting to match them with unmatched components of Old that are semantically equivalent but textually different. These components of New are classified as textual changes. Components of New that remain unmatched are classified as semantic changes.

Since textual changes are related to program text rather than program behavior, we only flag the semantic changes as different parts in the code. Once we obtain the equal and different parts in the code, we discard the differences and retain the equal parts of code. Feature similarity is then measured using a size metric. Size metrics are perhaps the most frequently used metrics in practice (Dagpinar and Jahnke 2003). The simplest and most commonly used size metric is lines of code (LOC) but it highly depends on coding style of programmers (Dagpinar and Jahnke 2003). There are other more advanced size metrics such as NIM (Number of Instance Methods) or TNOS (Total Number Of Statements). NIM counts the number of instance methods in a class, i.e., all public, protected and private methods defined in the interface of instances of a given class. The TNOS is a size metric that measures code size by counting the number of statements (e.g. for, if, return, switch, while) in each method. Since TNOS does not depend on the coding style of programmers and it is a significant predictor for the maintainability of software (Dagpinar and Jahnke 2003), we measure feature similarity in terms of the TNOS (Dit et al. 2013).

Following the Linked List example of Figure 4.4, the Similarity Comparison (part 4) compares the code of F2(PB) and F2(PA). As the code shows, the two methods are very similar, although they do not have the exact same code (the semantic changes being highlighted from lines 20 to 28 on product PB). It is reasonable for the code to differ, with PA implementing a list that is linked in a single fashion and PB implementing a list that is doubly linked. Even if the lists are sorted through the same bubble sort algorithm, said algorithm cannot be implemented in the exact same way considering the distinct number of links present between elements. From the example, it is possible to deduct that some the feature has been somehow modified since its PA implementation until its PB implementation. As a matter of fact, measuring the code, F2(PA) presents 6 statements while F2(PB) presents 7 statements. Taking in account that 4 of the 7 statements are equal, representing the same conditions in the code, the similarity percentage between F2(PA) and F2(PB) is around the 57%.

To sum up, our approach is applied to a model-based software products family with non-formalized variability. In the first stage, features from the products are identified at the model level. The second stage calculates all the possible isolation operations for the features. In the third stage, the code comparisons dictated by the calculated operations are executed to isolate the code of the features. Finally, the approach assesses the similarity degree between the features that appear in more than one product by performing a comparison in the fourth stage. The similarity between the features enables the classification of the clone-and-own relationships in one of the types described in Section 4.2 (Reimplemented, Modified, Adapted, Unaltered, or Ghost features).

4.4 Evaluation

This section presents the evaluation of our approach and the baseline, the description of the case studies where we applied the evaluation, the results obtained, the discussion, and the limitations. To evaluate the approach, we applied it to three long-living industrial case studies from two of our industrial partners: BSH, the leading manufacturer of home appliances in Europe; and CAF, an international provider of railway solutions all over the world.

4.4.1 Experimental Setup

The goals of this experiment are both measuring the performance of our approach in terms of precision and recall and comparing our approach with the baseline.

Figure 4.5 shows an overview of the process that was followed to evaluate our approach. The left part of the figure shows the input for the evalu-

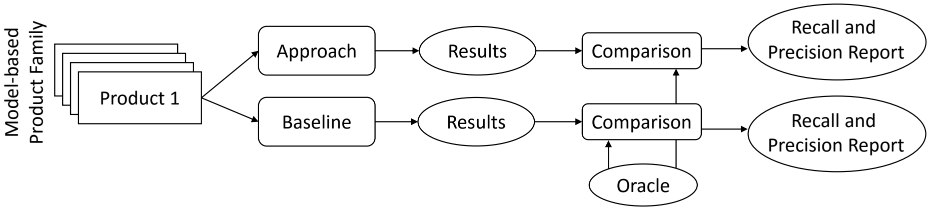


Figure 4.5: Evaluation process

ation process, provided by our industrial partners, which is the product family that has been specified through models. The product family is used to run our approach and the baseline. Although our industrial partners are not immune to the problem of knowledge vaporization (Ven et al. 2006), they provided us with documentation about some clone-and-own relationships and the identification of each relationship (reimplemented, modified, adapted, unaltered, or ghost). This documentation is used to build the oracle, which will be considered the ground truth and will be used to evaluate the results of our approach and the baseline.

The baseline is a previous version of our approach (Ballarin, Lapeña Martí, and Cetina 2016) that does not avoid the detection of irrelevant textual differences during the comparison of the source code of features (as described in Subsection 4.3.4). We compare the clone-and-own relationships obtained in both the baseline and our approach with the oracle in order to obtain precision and recall values.

Precision measures the number of elements from the solution (*SolutionCAO*) that are correct according to the the oracle (*OracleCAO*), and recall measures the number of elements of the solution (*SolutionCAO*) that are retrieved by the proposed solution (*OracleCAO*). A measure that combines both recall and precision is the harmonic mean of precision and recall, which is called the F-measure.

The recall and precision are calculated as follows:

$$Precision = \frac{SolutionCAO \cap OracleCAO}{SolutionCAO}$$

$$Recall = \frac{SolutionCAO \cap OracleCAO}{OracleCAO}$$

The F-measure that combines recall and precision is calculated as follows:

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

To calculate the precision and recall, we need to compute the true positives (TP); the number of elements in the solution that are actually correct according to the ground truth (the oracle), i.e., the clone-and-own relationships that are classified equal in both the solution and the ground truth ($SolutionCAO \cap OracleCAO$). The precision is calculated by dividing the TP by the total number of clone-and-own relationships in the solution (*SolutionCAO*). The recall is calculated by dividing the TP by the total number of clone-and-own relationships in the oracle (*OracleCAO*). In our case, each identified clone-and-own relationship that is present in both the results and the oracle will be a TP.

Precision values can range between 0% (which means that no single clone-and-own relationship of a given type from the results is present in the oracle) to 100% (which means that all the clone-and-own relationships of a given type from the results are present in the oracle).

Recall values can range between 0% (which means that no single clone-and-own relationship of a given type obtained from the oracle is present in the results) to 100% (which means that all the clone-and-own relationships of a given type from the oracle are present in the results). A value of 100% precision and 100% recall implies that both identifications are the same.

BSH: The Induction Hobs Domain

One of our industrial partners, the BSH group (`www.bsh-group.com`), has produced firmwares for their Induction Hobs (labeled under the Bosch and Siemens brands) for the last 15 years. The newest Induction Hobs (IHs) include the full cooking surface functionality, which calculates dynamic heating areas in an automatic fashion, activating or deactivating the areas depending on factors such as the utilized cookware shape, size, or position. In addition, more feedback is now provided to the user during the cooking process, including factors such as the exact cookware temperature, the temperature of the food being cooked, or real-time power consumption measurements. All of these changes have become possible through an increase of software complexity.

BSH provided us two case studies. The first case study entails a family of products that was specified using a Domain Specific Language (DSL) identified as IHDSL. After the specification, the IH's firmware was manually implemented (MI) in ANSI C by software engineers. Since this family of products belongs to BSH and uses manual implementation, we refer to this family as BSH-MI. Table 4.1 shows the characteristics of the BSH-MI case study. As the table shows, this family of products contains a total of 46 products and 81 features. In addition, Table 4.1 shows that the extracted oracle is composed by both 68 clone-and-own relationships that the features have across the different products of the family, and the identification of each clone-and-own relationship (e.g., 12 clone-and-own relationships are identified as reimplemented).

Table 4.1: Characteristics of the BSH-MI case study

Products	46
Features	81
Total clone-and-own relationships in the oracle	68
Reimplemented	12
Modified	19
Adapted	21
Unaltered	9
Ghost	7

The second case study entails a family of products that was also specified using IHDSL. After the specification, the IH's firmware was automati-

Table 4.2: Characteristics of the BSH-AI case study

Products	66
Features	47
Total clone-and-own relationships in the oracle	38
Reimplemented	0
Modified	8
Adapted	11
Unaltered	19
Ghost	0

cally implemented (AI) using M2T (model-to-text) transformation. This transformation was produced by Acceleo (Corredor et al. 2012). Since this family of products belongs to BSH and uses automatic implementation, we refer to this family as BSH-AI. Table 4.2 shows the characteristics of the BSH-AI case study, which has a total of 66 products and 47 features. The oracle has 38 clone-and-own relationships in total. These relationships are identified as modified (8 relationships), adapted (11 relationships), or unaltered (19 relationships).

CAF: The Train Control Domain

Our other industrial partner, CAF (`www.caf.net/en`), has produced a family of PLC software to control the trains that they have been manufacturing over more than 25 years. Their different kinds of trains (regular trains, subway, light rail, monorail, etc.) are installed all around the globe. Train units are geared with multiple pieces of equipment through their vehicles and cabins. Equipments come from different providers that design and manufacture them with the aim of carrying out specialized tasks in the train. Some examples are the traction equipment, the brake compressors, or the power-harvesting pantograph. The train unit is also equipped with control software, which is in charge of the cooperation of the installed equipments. The control software is created with two goals in mind: (1) orchestrating the equipments to achieve flawless train functionality, and (2) guaranteeing the compliance of the train unit with the prevalent regulations of the country where the train unit is to be installed.

The DSL of CAF has enough expressiveness to describe both the interactions between the main pieces of equipment installed in a train unit and the non-functional aspects related to regulation (such as signal quality or installed redundancy levels).

An example of the functionality that the DSL can specify is the coupling between train units. A train unit can physically connect to a second train unit and control it in order to increase its passenger capacity or to rescue the second train unit in case the former suffered any damage while functioning. After the specification using the DSL, the code is obtained by means of manual implementation (MI) in C. We refer to the family of products of this case study as CAF-MI.

Table 4.3 shows the characteristics of the CAF-MI case study. It has a total of 23 products and 121 features, whereas the oracle has 175 clone-and-own relationships in total. These relationships are identified as reimplemented (27), modified (23), adapted (78), unaltered (39), or ghost (8).

Table 4.3: Characteristics of the CAF-MI case study

Products	23
Features	121
Total clone-and-own relationships in the oracle	175
Reimplemented	27
Modified	23
Adapted	78
Unaltered	39
Ghost	8

4.4.2 Results

In this section, we present the results obtained for each case study in our approach and the baseline. Table 5.2 shows the values of precision, recall, and F-measure for each type of clone-and-own relationship (reimplemented, modified, adapted, unaltered and ghost) for the three case studies (BSH-MI, BSH-AI, and CAF-MI).

In both our approach and the baseline, the BSH-MI case study obtains the same results for precision and recall in the following relationships: reimplemented (88.89% of precision and 66.67% of recall), unaltered

Table 4.4: Values for Precision, Recall, and F-measure in the three case studies

		Precision		Recall		F-measure	
Case Study	Relationship	Approach	Baseline	Approach	Baseline	Approach	Baseline
BSH-MI	Reimplemented	88.89	88.89	66.67	66.67	76.19	76.19
BSH-MI	Modified	70.59	42.86	63.16	63.16	66.67	51.06
BSH-MI	Adapted	86.36	81.82	90.48	42.86	88.37	56.25
BSH-MI	Unaltered	87.50	87.50	77.78	77.78	82.35	82.35
BSH-MI	Ghost	100	100	42.86	42.86	60	60
BSH-AI	Reimplemented	-	-	-	-	-	-
BSH-AI	Modified	100	77.78	87.5	87.5	93.33	82.35
BSH-AI	Adapted	100	100	90.91	72.73	95.24	84.21
BSH-AI	Unaltered	100	100	89.47	89.47	94.44	94.44
BSH-AI	Ghost	-	-	-	-	-	-
CAF-MI	Reimplemented	72.73	72.73	88.89	88.89	80	80
CAF-MI	Modified	85.71	20.34	52.17	52.17	64.86	29.27
CAF-MI	Adapted	87.50	50	62.82	14.10	73.13	22
CAF-MI	Unaltered	66.67	69.23	41.03	23.08	50.79	34.62
CAF-MI	Ghost	100	100	75	75	85.71	85.71

(87.5% of precision and 77.78% of recall) and ghost (100% of precision and 42.86% of recall). Our approach reaches better results for precision in the modified and adapted relationships, and for recall in the adapted relationship (see the shaded cells in Table 5.2).

In the BSH-AI case study, our approach and the baseline obtain the same results for precision and recall in the unaltered relationship, whereas our approach outperforms the baseline for precision in the modified relationship and for recall in the adapted relationship.

In the CAF-MI case study, our approach and the baseline obtain the same results for precision and recall in the reimplemented and ghost relationships. The baseline outperforms our approach for precision in the unaltered relationship, whereas our approach outperforms the baseline for precision in the modified relationship, precision and recall in the adapted relationship, and recall in the unaltered relationship.

4.4.3 Discussion

The results show that there is no difference between our approach and the baseline for the reimplemented and ghost clone-and-own relationships. This is because our approach and the baseline use the same operations for the isolation of code fragments, so the code fragments used as input in the similarity step for the code comparison (which is different between our approach and the baseline) are the same in both approaches.

Reimplemented relationships are features that have been implemented from scratch, without using the source code as a template for the target code, and that have been, in some cases, developed by different teams of software engineers. Hence, the two codes that implement a reimplemented feature do not present common code. Since the two codes that implement a reimplemented feature do not share any code fragment, the result of the code comparison step is the same in both the approach and the baseline.

In case of the ghost relationships, since there is no code that implements the features, our approach and the baseline always coincide in the results.

In case of the unaltered, adapted, and modified relationships, our approach improves the results of the baseline. In both our approach and the baseline, once the source and target codes of a feature are isolated, they must be compared in order to determine which code fragments are equal between them. In case of unaltered features, incorrect code comparisons made by the baseline cause unaltered relationships to be incorrectly identified as adapted or modified relationships. In addition, the incorrect code comparisons made by the baseline cause adapted relationships to be incorrectly identified as modified. This makes the precision of the baseline worse with regard to our approach in all the case studies as follows: 27.73% for the modified relationship in the BSH manually implemented case study (BSH-MI), 22.22% for the modified relationship in the BSH automatically implemented case study (BSH-AI), and 65.37% for the modified relationship and 37.5% for the adapted relationship in the CAF manually implemented case study (CAF-MI).

Comparing the results of the automatically implemented case study with those of the two manually implemented case studies, it is possible to appreciate that the recall and precision values obtained are higher in the automatically implemented case study for both our approach and the baseline. In this scenario, the code is obtained by using a code generator, and in some cases, manually refined afterwards. Automatically generated code favors code comparisons in both our approach and the baseline because (1) the source and target code of a clone-and-own relationship that has not been manually refined should be identical and (2) when a human introduces code modifications, the refined code often uses the generated variables and methods.

4.4.4 Limitations

There are some limitations of our approach that must be acknowledged. To start with, some companies implement the code directly from requirement specifications, leading to families of software products implemented without the usage of models. Our approach, in its current state, is not applicable to said scenarios. The only stage of our approach that depends on models is the Model-based Feature Location. In order to adapt our approach to the mentioned circumstance, it would be necessary to develop techniques able to carry out feature location at the requisites level. In addition, if the features of each product are known beforehand, our approach could be adapted to start in Stage 2 (Feature Isolation).

Secondly, although in the evaluation we have chosen relationships for the oracle that can be isolated by our approach, depending on the products in the family of software products and on their particular configurations, it may not be possible for our approach to calculate all the isolation operations, or in other words, some features from some products may lack an isolation operation at the end of the execution of the algorithm. To solve this issue, our approach could suggest a selection of products with specific feature configurations designed to allow the algorithm to isolate non-isolated features. A software engineer could manually add these products to the family, enhancing the results of the algorithm.

Moreover, there is some degree of uncertainty associated with the disclosing of Clone-and-Own Relationships between products. In particular, the followed criteria is very rigid for the reimplementation and feature modification relationships. For instance, some results in reimplemented features could be incorrectly classified as modified features due to their low amounts of common code. The classification in these borderline cases is yet to be polished, and will be tackled in future works.

4.5 Related Work

The works related to the one presented can be found in two main knowledge areas: feature location at the code level, and feature location at the model level.

4.5.1 Feature Location at the Code Level

In this area, some works apply type systems to obtain relevant data when building the variability model. As an example, Typechef (Kästner, Giarrusso, et al. 2011) supplies an infrastructure to analyze variability through #ifdef directives. In (Kästner, Ostermann, and Erdweg 2012), the authors enhance Typechef so as to support variability at run-time.

Text similarity techniques build on mathematical methods to determine textual similarity. Latent Semantic Indexing (LSI) (Landauer and Psotka 2000) uses the number of occurrences in a set of words in large texts to obtain similarity measurements between features and source code, represented by Vector Space Models (VSM). These text similarity techniques have also been combined with dynamic analysis (Asadi et al. 2010).

Other works apply reverse engineering to source code in order to obtain variability models (Czarnecki and Wasowski 2007; She et al. 2011). In (Czarnecki and Wasowski 2007), propositional logic is used to describe dependencies between features. In (Nadi et al. 2014), Typechef and propositional logic are combined to extract conditions among features.

Program Dependence Analysis (PDA) is applied by several Feature Location approaches (Walkinshaw, Roper, and Wood 2007; Trifu 2009). PDA can be represented by Program Dependence Graphs (PDG), where nodes entail functions or global variables, and edges depict calls to functions or global variable accesses.

Trace analysis at run-time is used to define variability models through significant information. Upon execution, the technique produces traces that indicate which code has been run. Some authors (Eisenberg and Volder 2005) base their approaches on the analysis of the traces. In addition, other works mix dynamic and static analysis, such as LSI (Poshyvanyk et al. 2007), PDA (Eisenberg and Volder 2005) or VSM (Eaddy et al. 2008).

Apart from isolating the implementations of the features, our approach also extracts Clone-and-Own Relationships among features. The relationships can be utilized by software engineers to understand in a better manner the reuse patterns of said features, and to plan and propose reuse opportunities and improvements.

Other works enhance code reuse by comparing the source code of existing product variants. In (Fischer et al. 2014), associations between artifacts and their modules (i.e., features) are extracted to provide hints at what features could not be separated, or for which artifacts there are multiple order options available. In (Lin et al. 2017), recurring designs are detected in source code to extract templates as reuse opportunities. The templates can be managed and customized to generate code skeleton for the reusable features. This generated code skeleton contains semi-implemented code that is annotated with hints and comments of necessary modifications.

In contrast to the works mentioned above that take as input source code, our approach leverages models of different product variants. When companies such as our industrial partners use models as the main software artifact to develop software in the context of the Model-Driven Development (MDD) paradigm, it is necessary to provide feedback to developers at the model level. The works mentioned above such as (Fischer et al. 2014) and (Lin et al. 2017) do not consider the models, so their results are not applicable for MDD engineers. Instead, our approach considers the models, so the results are traced to the models and MDD engineers can make decisions at the model level, which is the main artifact in MDD.

4.5.2 Feature Location at the Model Level

In (Rubin and Chechik 2012), a framework for legacy product lines mining and automated refactoring is proposed. The authors contrast the input elements, matching those with a certain degree of similarity and merging them together. The work presented in (Zhang, Haugen, and Møller-Pedersen 2011) proposes an approach to automatically compare products, extracting their variability in terms of the Common Variability Language (CVL) (Haugen et al. 2008; Svendsen et al. 2010). In (Font, Arcega, et al. 2015), the authors present an approach for automating the formalization of variability in a given models family. The common and different parts of the models are specified as a set of placements over a base model and a library of model replacements. The ensuing Software Product Line (SPL) enables the derivation of new product models through the reuse of the extracted model fragments. Another approach can be found in (Martinez et al. 2015), where the authors propose comparisons to extract variability from all the possible kinds of assets. All the mentioned works target the formalization of the variability inherent to an SPL. Finally, (Font, Ballarin, et al. 2015) identifies model patterns in a models set, conceptualizing the obtained patterns as model fragments that can be reused.

All of these approaches are limited to finding model fragments that represent features, with the ultimate goal of formalizing the variability of a particular SPL. Opposite to said works, we built an approach that incorporates both feature location at the model level and comparisons at the code level, with the goal of isolating the implementations of individual features. In addition, our work discloses Clone-and-Own Relationships among the detected features. The relationships can be used by software engineers to suggest improvements and reuse opportunities, based on the knowledge about feature reuse that the relationships expose.

4.6 Conclusions

Identifying the clone-and-own relationships that are inherently present across a family of software products can help software engineers to suggest reuse improvements, to detect impediments, to analyze the cost-benefit payoffs of reuse against reimplementing, and to detect the maturity of the family.

In this paper, we have presented our approach to locate clone-and-own relationships between features in model-based families of software products. Our approach proposes an algorithm that retrieves the code associated with each feature by taking as input the information that the techniques on feature location provide. Next, our approach makes feature isolation at model and code level to obtain the source code of each feature in a particular product. Finally, our approach compares the code of the features that belong to more than one product by avoiding the detection of irrelevant textual differences in order to locate the clone-and-own relationships as Reimplemented, Modified, Adapted, Unaltered, or Ghost.

We have also shown the feasibility and generalization of our approach by applying it to real world environments of three industrial case studies in two different domains. We have successfully located the clone-and-own relationships presented in two product families of induction hob models, and in one product family of train control software. In the case of the induction hobs, one of the families had its code implemented manually and the other one, in an automatic way. In the case of the train control software, the product family had its code implemented manually.

When faced with unaltered, adapted, and modified relationships, our approach improves the results presented by the baseline. In the case of unaltered features, the baseline incorrectly classifies some of them as adapted or modified relationships. In the case of adapted relationships, the baseline sometimes makes incorrect classifications, flagging them as modified relationships. The precision of the baseline is worse than that of our approach in all the case studies: 27.73% for the modified relationship in the BSH manually implemented case study (BSH-MI), 22.22% for the modified relationship in the BSH automatically implemented case study (BSH-AI), and 65.37% for the modified relationship and 37.5% for the adapted relationship in the CAF manually implemented case study (CAF-MI).

Bibliography

Asadi, Fatemeh et al. (2010). "A Heuristic-Based Approach to Identify Concepts in Execution Traces". In: *14th European Conference on Software Maintenance and Reengineering, CSMR, March '10, Madrid, Spain.* Ed. by Rafael Capilla, Rudolf Ferenc, and Juan C Dueñas. IEEE Computer Society. ISBN: 978-0-7695-4321-5. DOI: `10.1109/CSMR.2010.17` (cit. on pp. 109, 131).

Ballarin, Manuel, Raúl Lapeña Martí, and Carlos Cetina (June 2016). "Leveraging Feature Location to Extract the Clone-and-Own Relationships of a Family of Software Products". In: pp. 215–230. DOI: `10.1007/978-3-319-35122-3_15` (cit. on pp. 111–113, 123).

Brambilla, Marco, Jordi Cabot, and Manuel Wimmer (2012). *Model-Driven Software Engineering in Practice.* 1st. Morgan & Claypool Publishers. ISBN: 1608458822, 9781608458820 (cit. on p. 109).

Corredor, Iván et al. (2012). "Model-Driven Methodology for Rapid Deployment of Smart Spaces Based on Resource-Oriented Architectures". In: *Sensors.* ISSN: 1424-8220. DOI: `10.3390/s120709286` (cit. on p. 126).

Czarnecki, Krzysztof and Andrzej Wasowski (2007). "Feature Diagrams and Logics: There and Back Again". In: *Software Product Lines, 11th International Conference, SPLC, Kyoto, Japan, September 10-14, 2007, Proceedings.* IEEE Computer Society. DOI: `10.1109/SPLINE.2007.24` (cit. on pp. 109, 131).

Dagpinar, M. and J. H. Jahnke (Nov. 2003). "Predicting maintainability with object-oriented metrics -an empirical comparison". In: *10th Working Conference on Reverse Engineering, 2003. WCRE 2003. Proceedings.* Pp. 155–164. DOI: `10.1109/WCRE.2003.1287246` (cit. on p. 121).

Dit, Bogdan et al. (Jan. 2013). “Feature location in source code: A taxonomy and survey”. In: *Journal of Software Maintenance and Evolution: Research and Practice* 25. DOI: 10.1002/smr.567 (cit. on pp. 116, 121).

Eaddy, Marc et al. (2008). “CERBERUS: Tracing Requirements to Source Code Using Information Retrieval, Dynamic Analysis, and Program Analysis”. In: *The 16th IEEE International Conference on Program Comprehension, ICPC, Amsterdam, The Netherlands, June 10-13, 2008*. Ed. by René L Krikhaar, Ralf Lämmel, and Chris Verhoef. IEEE Computer Society, pp. 53–62. DOI: 10.1109/ICPC.2008.39 (cit. on p. 132).

Eisenberg, Andrew David and Kris De Volder (2005). “Dynamic Feature Traces: Finding Features in Unfamiliar Code”. In: *21st IEEE International Conference on Software Maintenance (ICSM), 25-30 September 2005, Budapest, Hungary*. IEEE Computer Society, pp. 337–346. ISBN: 0-7695-2368-4. DOI: 10.1109/ICSM.2005.42 (cit. on p. 132).

Fischer, Stefan et al. (2014). “Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants”. In: *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*. ICSME ’14. Washington, DC, USA: IEEE Computer Society, pp. 391–400. ISBN: 978-1-4799-6146-7. DOI: 10.1109/ICSME.2014.61 (cit. on pp. 109, 132).

Font, Jaime, Lorena Arcega, et al. (2015). “Building Software Product Lines from Conceptualized Model Patterns”. In: *Proceedings of the 19th International Conference on Software Product Line*. SPLC ’15. New York, NY, USA: ACM. ISBN: 978-1-4503-3613-0 (cit. on pp. 109, 116, 117, 133).

Font, Jaime, Manuel Ballarin, et al. (July 2015). “Automating the variability formalization of a model family by means of common variability language”. In: pp. 411–418. DOI: 10.1145/2791060.2793678 (cit. on p. 133).

Haugen, Øystein et al. (2008). “Adding Standardized Variability to Domain Specific Languages”. In: *Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, 2008, Proceedings*. IEEE Computer Society, pp. 139–148. ISBN: 978-0-7695-3303-2. DOI: 10.1109/SPLC.2008.25 (cit. on p. 133).

Horwitz, Susan (June 1990). “Identifying the Semantic and Textual Differences Between Two Versions of a Program”. In: *SIGPLAN Not.* 25.6, pp. 234–245. ISSN: 0362-1340. DOI: `10.1145/93548.93574` (cit. on p. 120).

Kamiya, Toshihiro, Shinji Kusumoto, and Katsuro Inoue (2002). “CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code”. In: *IEEE Trans.* DOI: `10.1109/TSE.2002.1019480` (cit. on p. 119).

Kästner, Christian, Paolo G Giarrusso, et al. (2011). “Variability-aware parsing in the presence of lexical macros and conditional compilation”. In: *Proceedings of the 26th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2011.* Ed. by Cristina Videira Lopes and Kathleen Fisher. ACM. ISBN: 978-1-4503-0940-0 (cit. on p. 131).

Kästner, Christian, Klaus Ostermann, and Sebastian Erdweg (2012). “A variability-aware module system”. In: *Proceedings of the 27th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, USA, October 21-25, 2012.* Ed. by Gary T Leavens and Matthew B Dwyer. ACM. ISBN: 978-1-4503-1561-6 (cit. on p. 131).

Landauer, Thomas K and Joseph Psotka (2000). “Simulating Text Understanding for Educational Applications with Latent Semantic Analysis: Introduction to LSA”. In: *Interactive Learning Environments.* DOI: `10.1076/1049-4820(200008)8:2;1-B;FT073` (cit. on pp. 109, 131).

Li, Zhenmin et al. (2006). “CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code”. In: *IEEE Trans. Software Eng.* DOI: `10.1109/TSE.2006.28` (cit. on p. 119).

Lin, Y. et al. (Oct. 2017). “Mining implicit design templates for actionable code reuse”. In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE).* Vol. 00, pp. 394–404. DOI: `10.1109/ASE.2017.8115652` (cit. on pp. 109, 132).

Martinez, Jabier et al. (2015). “Bottom-up adoption of software product lines: a generic and extensible approach”. In: *Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA,* *July 20-24, 2015*. Ed. by Douglas C Schmidt. ACM. ISBN: 978-1-4503-3613-0 (cit. on pp. 109, 116, 117, 133).

Nadi, Sarah et al. (2014). "Mining configuration constraints: static analyses and empirical results". In: *36th International Conference on Software Engineering, ICSE 14, Hyderabad, India - May 31 - June 07, 2014*. Ed. by Pankaj Jalote, Lionel C Briand, and André van der Hoek. ACM, pp. 140–151. ISBN: 978-1-4503-2756-5. DOI: `10.1145/2568225.2568283` (cit. on p. 131).

Pham, Nam H. et al. (2012). "Clone Management for Evolving Software". In: *IEEE Transactions on Software Engineering* 38.undefined, pp. 1008–1026. ISSN: 0098-5589. DOI: `doi.ieeecomputersociety.org/10.1109/TSE.2011.90` (cit. on p. 109).

Poshyvanyk, Denys et al. (2007). "Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval". In: *IEEE Trans. Software Eng.* DOI: `10.1109/TSE.2007.1016` (cit. on p. 132).

Rubin, Julia and Marsha Chechik (2012). "Combining Related Products into Product Lines". In: *Fundamental Approaches to Software Engineering - 15th International Conference, FASE 2012 Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. Ed. by Juan de Lara and Andrea Zisman. Vol. 7212. Lecture Notes in Computer Science. Springer, pp. 285–300. ISBN: 978-3-642-28871-5 (cit. on pp. 109, 116, 117, 133).

Selic, Bran (2003). "The Pragmatics of Model-Driven Development". In: *IEEE Software*. DOI: `10.1109/MS.2003.1231146` (cit. on p. 114).

She, Steven et al. (2011). "Reverse engineering feature models". In: *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*. Ed. by Richard N Taylor, Harald C Gall, and Nenad Medvidovic. ACM. ISBN: 978-1-4503-0445-0. DOI: `10.1145/1985793.1985856` (cit. on pp. 109, 131).

Svendsen, Andreas et al. (2010). "Developing a Software Product Line for Train Control: A Case Study of CVL". In: *Software Product Lines: Going Beyond - 14th International Conference, SPLC, Jeju Island, South Korea,* *September 13-17, 2010. Proceedings*. Ed. by Jan Bosch and Jaejoon Lee. Vol. 6287. Lecture Notes in Computer Science. Springer, pp. 106–120. ISBN: 978-3-642-15578-9 (cit. on p. 133).

Trifu, Mircea (2009). "Improving the Dataflow-Based Concern Identification Approach". In: *13th European Conference on Software Maintenance and Reengineering, CSMR 2009, Architecture-Centric Maintenance of Large-SCale Software Systems, Kaiserslautern, Germany, 24-27 March 2009*. Ed. by Andreas Winter, Rudolf Ferenc, and Jens Knodel. IEEE Computer Society. ISBN: 978-0-7695-3589-0. DOI: `10.1109/CSMR.2009.34` (cit. on p. 132).

Ven, J. et al. (2006). "Design decisions: The bridge between rationale and architecture". In: vol. Rationale Management in Software Engineering. Springer Berlin Heidelberg, pp. 329–348 (cit. on p. 123).

Walkinshaw, Neil, Marc Roper, and Murray Wood (2007). "Feature Location and Extraction using Landmarks and Barriers". In: *23rd IEEE International Conference on Software Maintenance (ICSM 2007), October 2-5, 2007, Paris, France*. IEEE. ISBN: 978-1-4244-1256-3. DOI: `10.1109/ICSM.2007.4362618` (cit. on p. 132).

Zhang, Xiaorui, Øystein Haugen, and Birger Møller-Pedersen (2011). "Model Comparison to Synthesize a Model-Driven Software Product Line". In: *Software Product Lines - 15th International Conference, SPLC, Munich, Germany, August 22-26, 2011*. Ed. by Eduardo Santana de Almeida et al. IEEE, pp. 90–99. ISBN: 978-1-4577-1029-2 (cit. on pp. 109, 116, 117, 133).

Chapter 5

Measures to report the Location Problem of Model Fragment Location

Model Fragment Location (MFL) aims at identifying model elements that are relevant to a requirement, feature, or bug. Many MFL approaches have been introduced in the last few years to address the identification of the model elements that correspond to a specific functionality. However, there is a lack of detail when the measurements about the search space (models) and the measurements about the solution to be found (model fragment) are reported. Generally, the only reported measure is the model size. In this paper, we propose using five measurements (size, volume, density, multiplicity, and dispersion) to report the location problems. These measurements are the result of analyzing 1,308 MFLs in a family of industrial models over the last four years. Using two MFL approaches, we emphasize the importance of these measurements in order to compare results. Our work not only proposes improving the reporting of the location problem, but it also provides real measurements of location problems that are useful to other researchers in the design of synthetic location problems.

5.1 Introduction

From the timeless traceability activity (Winkler and Pilgrim 2010) to recent research efforts on Feature Location (J. Martinez et al. 2015), (Jaime Font et al. 2016), (J. Font et al. 2017) and Bug Location (Arcega, Jaime Font, Oystein Haugen, et al. 2017), Model Fragment Location (MFL) has been gaining momentum. Overall, these MFL approaches address the identification of the model elements that are relevant to a requirement, feature, or bug.

Current MFL approaches have leveraged Information Retrieval, Linguistic techniques, and Search-based techniques to achieve the location of relevant model fragments. These approaches provide the algorithms and the parameters used to tune them in detail. Nonetheless, there is a lack of detail when the measurements about the search space (models) and the measurements about the solution (model fragment) are reported. Generally, the only reported measure is the model size. However, in most of the cases, the model-size values are not comparable among different works since different models are measured in different ways.

In this paper, we propose using five measurements (size, volume, density, multiplicity, and dispersion) to report the location problems during MFL. On the one hand, size and volume measure the search space. On the other hand, density, multiplicity, and dispersion measure the solution to be located. Our proposed measures are the result of analyzing 1,308 MFLs performed over the last four years in models of the industrial dimensions of CAF [1].

Properly reporting the location problem is important because otherwise it is not possible to compare the performance of different approaches with each other. It is not the same challenge to locate a large model fragment in a small model than to locate a small and scattered model fragment over several large models. We illustrate this phenomenon by comparing the performance of two MFL approaches in terms of precision and recall, which are performance measures that are widely used by the research community.

[1] http://www.caf.net/en

We not only proposes improving the reporting of the location problem, but we also provide real measures of location problems during MFL on industrial models. The aim of these values is twofold: (1) to provide researchers who create synthetic location problems with a reference from real-world problems and (2) to raise awareness among researchers of MFL approaches regarding the profile of real-world MFL problems.

The remainder of the paper is structured as follows. Section 5.2 provides an overview of MFL. Section 5.3 clarifies the way to count model elements and introduces our measurements for the location problem. Section 5.4 presents the values in the CAF case study. Section 5.5 performs a statistical analysis to provide evidence of the significance of the results. Section 5.6 discusses the outcomes of the paper. Section 5.7 discusses the works that are related to our work. Finally, Section 5.8 concludes the paper.

5.2 Overview of Model Fragment Location

Traceability Links Recovery (TLR) is one of the most common activities performed during the software system maintenance phase. TLR is concerned with establishing the model fragment that implements a specific natural language requirement. TLR among requirements and models is one type of MFL. Following, we illustrate MFL using TLR.

Figure 5.1 shows an excerpt taken from a real-world train of a Train Control and Management Language (TCML) model. TCML is the domain-specific language that is used by our industrial partner and was designed following UML conventions. TCML has the expressiveness required to describe both the iteration between the main pieces of equipment installed in a train unit and the non-functional aspects related to regulation.

For TLR, the input query is a natural language requirement for which the model fragment must be retrieved. The example depicted in Figure 5.1 shows a natural language requirement describing the high voltage functionality of one of the projects of our industrial partner. This requirement is as follows: "The PLC will command the raise of panto-

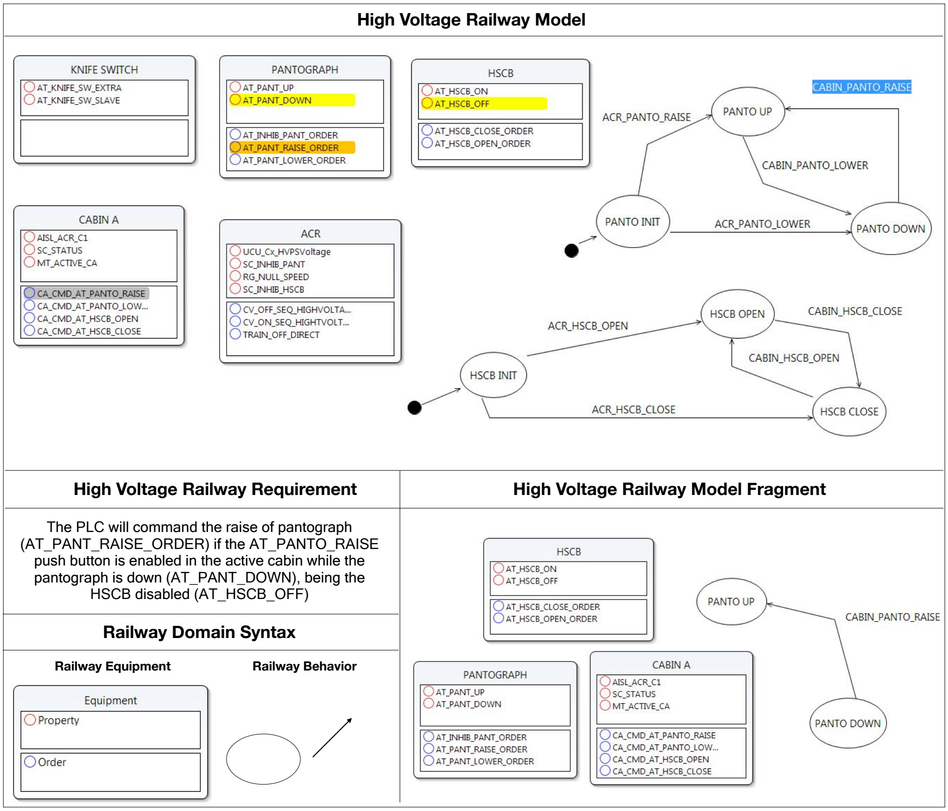


Figure 5.1: Example of a CAF model and model fragment

graph (AT_PANT_RAISE_ORDER) if the AT_PANTO_RAISE push button is enabled in the active cabin while the pantograph is down (AT_PANT_DOWN), being the HSCB disabled (AT_HSCB_OFF)".

In TLR, as in other MFL activities such as feature location or bug location, their is a search space (a model defining the set of all possible solutions) and a solution (model elements to be found). Focusing on the example shown in Figure 5.1, the search space corresponds to the model (see the top part of the figure) and the solution corresponds to the model fragment (see the bottom right part of the figure).

The top part of Figure 5.1 shows a train unit that is furnished with multiple pieces of equipment. The equipment shown includes: Knife switch, Cabin A, Pantograph, ACR, and HSCB. Each piece of equipment (e.g., PANTOGRAPH) in a train unit has a collection of properties (e.g., AT_PANT_UP) and a collection of orders (e.g., AT_INHIB_PANT_ORDER). The communications among the pieces of equipment are addressed through the state machines. For instance, to go from a pantograph in the down state to a pantograph in the up state, the transition CABIN_PANTO_RAISE is triggered.

The bottom right part of Figure 5.1 shows the model fragment to be located. The model fragment is comprised of several model elements including the *HSCB* and its property, the *PANTOGRAPH* with its corresponding property and order, and the *CABIN A* with its order. Also, the model fragment comprises a part of a state machine that represents the functionality associated with the raising of the pantograph.

5.3 Counting Model Elements

In this paper, we propose using five measurements (size, volume, density, multiplicity, and dispersion) to report the location problems. The count of model elements is important in these measurements. Currently, there are several approaches that are in charge of carrying out this task. [21].

C1 Counting the number of elements in a model: This approach is computed as the number of elements of the model. However, this approach neglects both the complexity of the elements and the differences among diagrams.

C2 Weight factors per model element: This approach measures the size of a model taking into account the complexity of the elements because the complexity and information that is provided by the diagram elements are not the same for all of the elements. Based on the findings of (Koffka 2013), (Störrle 2014) proposes defining complexity levels (e.g., simple, medium, and large) where each complexity level is related to a weight (e.g., simple = 1, medium = 1.5, large = 2). From these weights, this approach is computed as the number of elements weighted by complexity. Unfortunately, this approach neglects the inherent differences among diagrams.

C3 Weight factors per model element per diagram type: This approach measures the size of a model taking into account not only the complexity of the elements but also the differences among diagrams. To do this, in (Störrle 2014), the weight of each diagram element is calculated as $weight(e) = log_2(|E|)$, where e is the element whose weight is being calculated and E is the class of the element according to the diagram.

After applying these three approaches to a set of models, (Störrle 2014) came to the conclusion that their results are extremely correlated. In fact, none of these approaches yields significant better results than the other ones. Therefore, even though any of these approaches can be used to count model elements, C1 is strongly recommended since it is the easiest to implement and compute.

Based on the above, we propose reporting the size of models by means of C1. However, the size of a model is not enough to measure the search space and the solution in MFL. In the following section, we propose measurements to report the location problem.

5.3.1 Measurements for Model fragment Location

This section presents the measurements that we propose for reporting the location problem of MFL. Some of these measurements focus on measuring the search space, and the rest of them focus on measuring the solution to be searched for. Figure 6.3 shows the measurements of the search space (Size, and Volume) and the measurements of the solution (Density, Multiplicity, and Dispersion).

In order to measure the search space, we propose the following two measurements:

- **Size** measures the number of elements that the model contains from the metric C1. Since the larger the model, the larger the search space, this measurement determines how complex the search space ends up being in order to find the solution.

The first row of Figure 6.3 represents this measurement conceptually, where a solution is searched for in two different models. The first model is smaller than the second one, so the search space for the first model is smaller than the search space for the second model.

Given the example of Figure 5.1, the model is composed of five pieces of equipment (knife switch, pantograph, HCSB, cabin, and ACR). Moreover, the state diagrams of the model contain six states and eight transactions. Since the size of the model is computed as the sum of all of these elements, the size is 19 model elements. This measurement is frequently used in most MFL works. However, event through this measurement is not a novelty for our work, we include it for the sake of completeness.

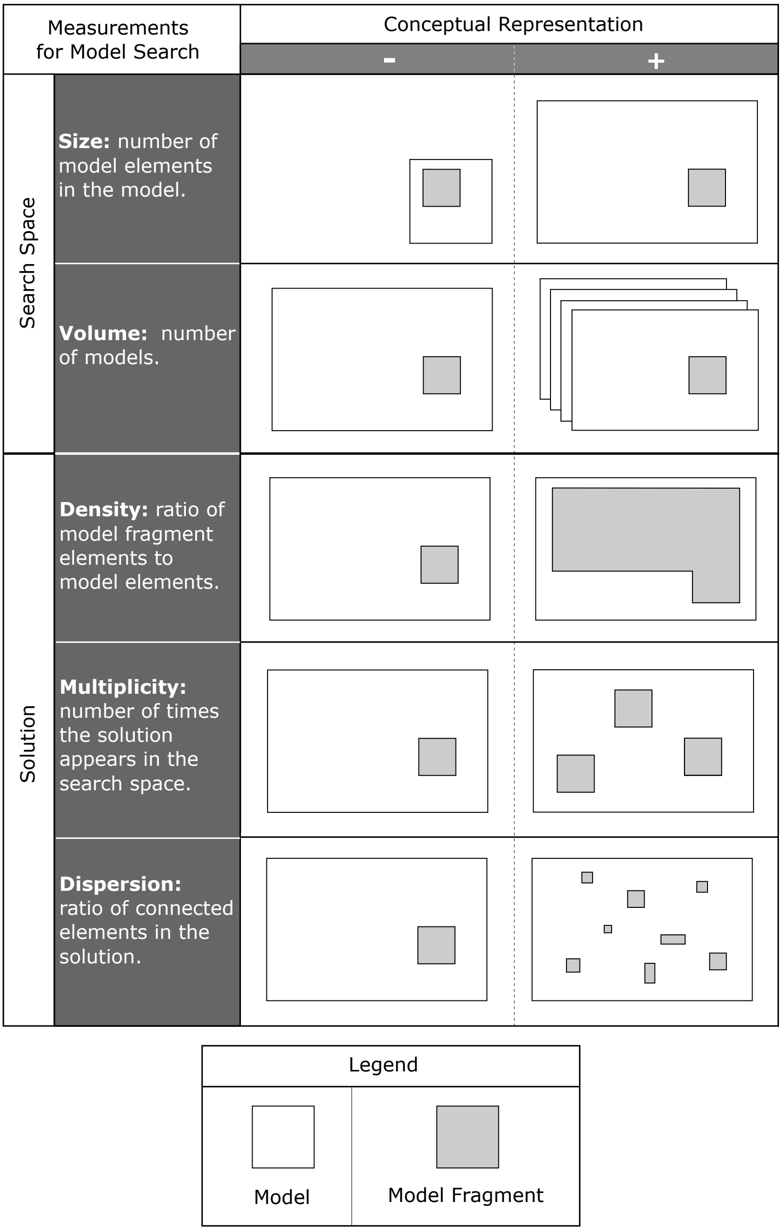
- **Volume** measures the number of models that compose the search space where a solution is searched for. Since the larger the number of models, the larger the search space, this measurement determines how large the search space becomes based on the number of models.

  The second row of Figure 6.3 represents this measurement conceptually, where a solution is searched for in two different search spaces. The first search space is composed of a single model; in contrast, the second search space is composed of several models.

  In the example of Figure 5.1, there is only one model. Since the search space is only composed of one model, the volume in this example is equal to one model.

In order to measure the solution, we propose the following three measurements:

- **Density** measures the percentage of model elements that realize a solution. In other words, since the model fragment is composed of



Size: number of model elements in the model.

Volume: number of models.

Density: ratio of model fragment elements to model elements.

Multiplicity: number of times the solution appears in the search space.

Dispersion: ratio of connected elements in the solution.

Figure 5.2: Conceptual representations of the measurements

the model elements that realize the solution, the density is computed as the ratio of model fragment elements to model elements. Since the larger the model fragment, the larger the density, this measurement determines how large the solution ends up being with respect to the model.

The third row of Figure 6.3 represents this measurement conceptually, where a model realizes two different solutions. The first solution is realized by a model fragment that contains a few model elements; in contrast, the second solution is realized by a model fragment that contains the majority of the model elements.

In the example of Figure 5.1, the size of the model is equal to 19, and the size of the model fragment can be computed as the number of elements that are highlighted, therefore the size of the model fragment is equal to six. The density is equal to 31.57% meaning that the model elements are part of the solution model fragment.

- **Multiplicity** measures the number of times the solution appears in the search space. Since the more solutions found, the greater the multiplicity, this measurement determines how complex the search ends up being, based on the number of solutions that the search space contains for the same solution.

  The fourth row of Figure 6.3 represents this measurement conceptually, where a solution is searched for in two models. The search in the first model reveals one model fragment as the solution. In contrast, the search in the second model reveals three model fragments, so there are three solutions in the model.

  In the example of Figure 5.1, none of the model elements are repeated, so it is not possible to find two model fragments with the same elements. A bigger model may contain more complex state machines, so it might be possible to find patterns that are repeated. However, in the Figure 5.1, none of the model elements are repeated, so the multiplicity is equal to 1.

- **Dispersion** measures the ratio of connected elements in the solution. Specifically, a model fragment is composed of the model elements that realize a solution, but these elements may or may not be connected in the model. Therefore, the elements of the model fragment can be divided into different groups in the model. Dispersion is computed as the ratio between the number of groups and the number of elements. Its value is from 0 to 1, where values around 0 indicate a strong connection among the solution elements and values around 1 indicate a strong dispersion among the solution elements. Since the more groups found, the larger the dispersion, this measurement determines how complex the search ends up being depending on whether or not the model elements that compose a model fragment are linked.

  The fifth row of Figure 6.3 represents this measurement conceptually, where a solution is realized by two models. In the first model, all of the elements that realize the solution are linked, so the model fragment is a unity. However, in the second model, the elements that realize the solution are disconnected, so the model fragment is divided into groups where each group is composed of one or more model elements.

  In the example of Figure 5.1, none of the elements of the model fragment are connected to the others, so there are as many groups as number of elements in the model fragment. Since the model fragment is composed of the elements that are highlighted, the model fragment contains six elements and the number of groups is equal to six. Therefore, when the dispersion is computed as the ratio between the number of groups and the number of elements, we obtain a dispersion value that is equal to 1 this means there is a strong dispersion among the model fragment elements.

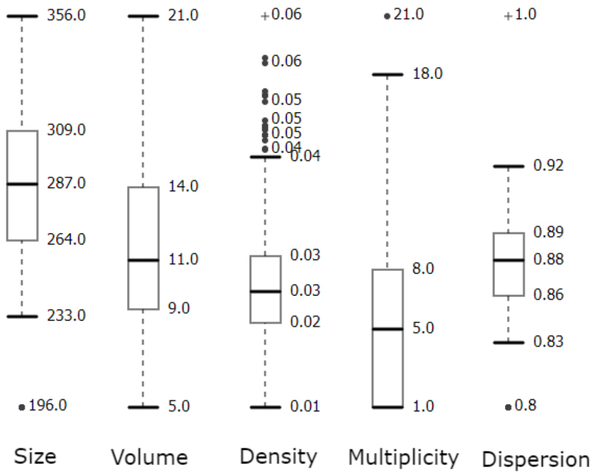


Figure 5.3: Maximum, Minimum, and Mean values of the Case study Measurements

5.4 Values in the CAF Case Study

This section presents the resulting values after applying MFL on the industrial models of CAF as part of their migration process between their model set to a Model-based Software Product Line. We analyze the result of applying 1,308 MFLs. We provide the results in Figure 5.3 and Table 5.1 as reference from real-world MFL problems.

Figure 5.3 depicts a box plot with the values including the maximum, the minimum, and the mean of the values for each of the proposed measurements. Each column in the plot corresponds to one of the proposed measurements (correspondingly named at the bottom of the plot). The results obtained are as follows:

- The **Size** measurement of the search space is between 196.0 model elements and 356.0 model elements. In other words, the largest search space used in the case study is composed of 356 model elements. In contrast, the smallest search space used is composed of a total of 196 model elements. In addition, most MFLs were performed in search spaces with sizes between 264 model elements and 309 model elements, with the search space with a size of 272 model elements being the most frequent.

- The **Volume** measurement of the search spaces is between 5 models and 21 models. This means that the search spaces where the MFL approach were performed are between these values. Most MFLs were performed in search spaces with a volume between 9 models and 14 models, with 10 models being the most common volume among all of the search spaces.
- The **Density** measurement of the solutions is between 1% and 6%, with the most common density interval being between 2% model elements and 3% model elements. In the case study, the most repeated solution has a density of 2% model elements.
- The **Multiplicity** measurement of the solutions is between 1 time and 21 times, with the most repeated range for each of the solutions oscillating between 1 time and 8 times. This means that, given a solution, the total of times that the solution can be located is between 1 time and 8 times as maximum. The most frequent multiplicity of our solutions has a value of 1 times.
- The **Dispersion** measurement of the solutions is between 0.8 groups / elements and 1.0 groups / elements. The range is between 0.83 groups / elements and 0.92 groups / elements. This means that the total of connected elements is between of these values. In the case study, the most repeated number of connected model elements for each solution is 0.85 groups / elements.

It is important to emphasize that the MFL was not performed on all of the existing models of our industrial partner. Domain experts were involved during the MFL approach, contributing their domain knowledge. A way to contribute with this domain knowledge is by restricting the location approach to those models in which the domain experts estimate that the solution will be found (in the case of the initialization of the Software Product Lines, i.e., the features).

Table 5.1: TOP 10 most frequent obtained results during Model Fragment Location in Industrial Models

	Search Frequency									
	Size		**Volume**		**Density**		**Multiplicity**		**Dispersion**	
	Value	MFLs	Value	MFLs	Value	MFLs	Value	MFLs	Value	MFLs
1	272	82	10	240	0.0267	34	1	608	0.857	425
2	321	82	9	189	0.0257	29	7	129	0.889	229
3	262	80	11	176	0.028	29	5	128	0.875	200
4	287	73	14	168	0.0249	28	6	111	0.833	162
5	279	48	8	104	0.0218	27	8	87	1.0	114
6	281	44	12	96	0.0297	25	9	74	0.9	60
7	322	43	16	64	0.03	25	10	51	0.8	53
8	292	43	13	52	0.0197	24	11	32	0.909	50
9	233	42	15	45	0.0265	21	13	27	0.923	10
10	332	41	7	42	0.0244	21	12	25	0.917	5

Table 5.1 shows the relationship between the value obtained from a specific measurement and the number of MFLs that use this value. In other words, Table 5.1 presents an overview of the top frequently MFLs carried out on the industrial models. Each column in the table identifies one of the proposed measurements in this work. The columns below these correspond to the most frequently obtained value (named Value in Table 5.1) regarding the number of MFLs.

According to the values shown in the table, 272 model elements are the most frequent *Size* of the search spaces in the case study, with a total of 82 MFLs carried out. According to *Volume*, the most frequent value is 10, meaning that 10 is the number of models that compose the search space where a model fragment is located.

With regard to the solution measurements (*Density*, *Multiplicity*, and *Dispersion*), 2.67 % is the most common percentage of model elements that realize the solutions, which was obtained in a total of 34 MFLs. Also, 1 time is the most common *Multiplicity* obtained during 608 MFLs, and 0.857 is the most frequent ratio of groups/elements, obtained during 425 MFLs.

5.4.1 Experimental Setup

The goal of this experiment is to determine if MFL is influenced by the measurements presented. In other words, this experiment empowers us to determine whether the results of MFL depend on the values of measurements or MFL obtains the same results whatever the values of the measurements are. To do this, the experiment addressed the searches in the models of the CAF Case Study by means of two approaches that obtain the best results to recover Traceability between requirements and models (Winkler and Pilgrim 2010). The first one (Spanoudakis et al. 2004) is a Linguistic Rule-Based (Linguistic) approach that is based on Parts-of-Speech (POS) Tagging and Traceability rules. The second one (De Lucia et al. 2004) is an Information Retrieval (IR) approach that is based on Latent Semantic Indexing (LSI) and Singular Value Decomposition (SVD). The results were analyzed based on the following measurements:

- Size is already studied in other research, in fact, this measurement is frequently used in most MFL works. Therefore, although this measurement is included in our research for the sake of completeness, we did not evaluate the relevance of this measurement in this work.

- Volume measures the number of models. Therefore, to evaluate the impact of this measurement, we took into account searches where the search space is composed of one or more models and only one of them contained the solution. Moreover, the search spaces had to have models with a similar size, the solutions had to have similar density values and dispersion values, and the multiplicity values had to be equal to one, which means that there is only one solution in the search space. Therefore, the values for the other measurements (Size, Density, Multiplicity, and Dispersion) are similar or equal for all of the searches.

- Density measures the ratio of model fragment elements to model elements. Therefore, to evaluate the impact of this measurement, we took into account searches where the search space was composed of the same model and the solutions had different numbers of elements so that the density was not the same for all of the searches.

Given that the search space was the same for all of the searches, the size and the volume values were the same for all of the searches. Moreover, the solutions had to have similar multiplicity and dispersion values. Therefore, the values for the other measurements (Size, Volume, Multiplicity, and Dispersion) are similar or equal for of the searches.

- Multiplicity measures the number of times the solution appears in the search space. Therefore, to evaluate the impact of this measurement, we took into account searches where each search space contains a different model and the searched solution is the same. Moreover, the search spaces had to have similar volume values and size values, and the solutions had to have similar density values and similar dispersion values. Therefore, the values for the other measurements (Size, Volume, Density, and Dispersion) are similar or equal for all of the searches.
- Dispersion measures the ratio of connected elements in the solution. Therefore, to evaluate the impact of this measurement, we took into account searches where the search space was composed of the same model and the elements of the solutions were scattered in the model to a greater or lesser extent. Thus, the dispersion values were not the same for all the searches. Given that the search space was the same for all of the searches, the size and the volume values were the same for all of the searches. Moreover, the solutions had to have similar density values and multiplicity values. Therefore, the values for the other measurements (Size, Volume, Density, and Multiplicity) are similar or equal for all of the searches.

For each search, each approach generates a model fragment as a possible solution. Then, we compare this possible solution with the real solution, which we know beforehand based on the CAF Case Study. Once the comparison is performed, a confusion matrix is calculated. Therefore, we obtain two confusion matrices, one for the Linguistic approach and one for the IR approach.

A confusion matrix is a table that is often used to describe the performance of a classification model (in this case, the approaches) on a set of test data (the resulting model fragments) for which the true values are known (from the CAF Case Study). In our case, each solution that is generated by the approaches is a model fragment that is composed of a subset of the model elements that are part of the model (where the solution is being searched for). 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 create a report that includes four performance metrics (precision, recall, the F-measure, and MCC) for each of the searches for each approach.

Precision measures the number of elements from the solution that are correct according to the ground truth (the solutions in the CAF Case Study) and is defined as follows:

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

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

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

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

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

However, none of these previous measures correctly handle negative examples (TN). The **Matthews Correlation Coefficient (MCC)** is a correlation coefficient between the observed and predicted binary classifications that takes into account all of the observed values (TP, TN, FP, FN) and is defined as follows:

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

Recall values can range between 0% (which means that no single model element from the solution from the CAF Case Study is present in the model fragment of the obtained solution) to 100% (which means that all the model elements from the CAF Case Study are present in the obtained solution). Precision values can range between 0% (which means that no single model element from the obtained solution is present in the solution from CAF Case Study) to 100% (which means that all of the model elements from the obtained solution are present in the solution from the CAF Case Study). A value of 100% precision and 100% recall implies that both the obtained solution and the solution from the oracle are the same. MCC values can range between -1 (which means that there is no correlation between the prediction and the solution) to 1 (which means that the prediction is perfect). Moreover, a MCC value of 0 corresponds to a random prediction.

Table 5.2: Mean Values and Standard Deviations for Precision, Recall and the F-Measure for the approaches depending on the measurement evaluated

		Precision	Recall	F-Measure	MCC
Volume	Linguistic	26.84 ± 18.78	50.75 ± 17.12	30.46 ± 17.72	0.32 ± 0.16
	IR	34.52 ± 27.29	34.19 ± 20.26	27.28 ± 12.57	0.29 ± 0.13
Density	Linguistic	22.96 ± 22.92	43.90 ± 29.45	25.89 ± 22.25	0.25 ± 0.23
	IR	66.22 ± 27.19	48.08 ± 18.85	52.09 ± 18.46	0.53 ± 0.18
Multiplicity	Linguistic	68.43 ± 21.06	55.56 ± 0.00	60.15 ± 7.96	0.60 ± 0.10
	IR	77.08 ± 25.66	18.06 ± 8.27	27.41 ± 9.30	0.35 ± 0.08
Dispersion	Linguistic	23.50 ± 21.86	48.48 ± 25.96	27.25 ± 20.59	0.27 ± 0.21
	IR	66.41 ± 28.25	43.47 ± 19.57	49.28 ± 19.61	0.51 ± 0.20

5.4.2 Results

In Table 5.2, we outline the results aggregated for each of the approaches. Each row shows the Precision, Recall, the F-measure, and MCC obtained by each approach taking into account the selected searches for each new measurement.

The IR approach achieved the best precision results for all of the measurements. However, the best results for the other performance indicators depend on which measurement is evaluated. According to Density, the IR approach achieved the best results for all of the performance indicators, providing a mean precision value of 66.22%, a recall value of 48.08%, a combined F-measure value of 52.09%, and a MCC value of 0.53. In contrast, according to Multiplicity and Volume, the Linguistic approach achieved the best results for recall, the F-measure, and MCC. Moreover, both approaches achieved the best precision results for searches that evaluate Multiplicity, providing mean precision values up to 68%.

5.5 Statistical Analysis

To properly compare the different searches, the data resulting from the empirical analysis was analyzed using statistical methods.

5.5.1 Statistical Significance

A statistical test must be run to assess whether there is enough empirical evidence to claim that the new measurements have an impact on the results. Therefore, we have to consider their values to search in models. To achieve this, two hypotheses for each new measurement are defined: the null hypothesis H_0, and the alternative hypothesis H_1. The null hypothesis H_0 is typically defined to state that there is no impact on the searches even though the measurements have different values. The alternative hypothesis H_1 states that there is an impact on the searches depending on whether the measurement values differ. In such a case, a statistical test aims to verify whether the null hypothesis H_0 should be rejected.

The statistical tests provide a probability value, $p - Value$. The $p - Value$ obtains values between 0 and 1. The lower the $p - Value$ of a test, the more likely that the null hypothesis is false. It is accepted by the research community that a $p{-}Value$ under 0.05 is statistically significant (Arcuri and Briand 2014), and so the hypothesis H_0 can be considered false.

The test carried out depends on the properties of the data. Since our data does not follow a normal distribution in general, our analysis requires the use of nonparametric techniques. There are several tests for analyzing this kind of data; however, the Quade test is the most powerful when working with real data (García et al. 2010).

Table 6.6 shows the Quade test statistic and $p{-}Values$ for recall and precision. The values that are pointed out indicate which ones are statistically significant, so the hypothesis H_0 can be considered false. At least one of the values for recall or precision for one of the approaches is statistically significant. Consequently, we can state that the new measurements have an impact on the searches, although the degree of this impact is an issue that remains as future work.

Table 5.3: Quade test statistic and $p - Values$

			Precision	Recall
Volume	Linguistic	p-Value	**0.013**	0.629
		Statistic	**18.00**	0.27
	IR	p-Value	**0.013**	**0.031**
		Statistic	**18.00**	**10.59**
Density	Linguistic	p-Value	0.561	**0.024**
		Statistic	0.62	**0.024**
	IR	p-Value	**0.020**	**0.047**
		Statistic	**6.58**	**4.59**
Multiplicity	Linguistic	p-Value	0.882	0.125
		Statistic	0.02	NaN*
	IR	p-Value	0.769	**0.015**
		Statistic	0.10	**25**
Dispersion	Linguistic	p-Value	0.220	0.700
		Statistic	1.63	0.15
	IR	p-Value	**0.003**	0.451
		Statistic	**12.03**	0.60

*The recall values are equal for all of the searches in this case.

5.6 Discussion

The results highlight that the proposed measurements have an impact on the outcomes of the studied approaches. In this work, we have identified a series of facts that serve as a starting point for discussing why the proposed measurements and the provided values are significant for the research community. These facts are discussed in the following paragraphs:

1 From the results, it is possible to conclude that size reports do not provide enough information on the search problem. This is due to the inability of the size measurement to accurately represent the inherent challenge levels of models.

This issue is better illustrated through an example taken from the case study. In the example, there are two models of sizes 37 and 113 elements (respectively) and a feature that is present in both models. At first glance, one may expect it to be easier to find the feature in the smaller model. However, in the first model, the feature is implemented by a model fragment that contains only two elements, and all of the elements in the model contain similar texts and word patterns. In the second model, the feature is implemented by 28 elements, which are clearly differentiated from the rest of the elements in the model in terms of text and word patterns. In the depicted scenario, both approaches are able to find the second model fragment with much greater accuracy than the first model fragment, rendering the size of the model insufficient to depict the search problem.

2 Search problems in models are relatively new when compared to search problems in other kinds of documents such as the web or source code. Hence, search problems in models have inherited measurements that are accepted by the community in similar fields. However, the novelty of applying searches to models is the models, and it is not possible to directly apply measurements to models from webs and code. In that sense, our work puts into perspective that this novel search problem requires more attention on how to report it in a correct manner so that performance results can be properly evaluated.

3 Due to the absence of real work datasets, synthetic datasets are very common and popular in the research community. These synthetic datasets are useful in testing extreme scenarios and situations. Since our findings show that the proposed measurements are significant and impactful with regard to the performance of search approaches, we provide their real-world values so that designers can carry out a real-world, search-problem profiling process when designing synthetic test cases and extreme search-problem scenarios. Therefore, the reference values for the measurements that have been presented in this paper can be useful for other researchers.

4 The results clearly show that the defined measurements have an impact on the results of the approaches. However, it may be possible to define new measurements or derive other measurements that are from the ones presented. Moreover, it may be possible to try and identify patterns in the results of the approaches based on the values of the measurements. Finally, we have not studied how different approaches are affected by each measurement; in other words, some approaches may be more sensitive to the values of certain measurements while not being affected by other measurements. All of these possibilities for further exploration of the measurements and their impact remain as future work.

In summary, the results of our work are promising and open the door for the study of the particularities of reporting the location problem of MFL.

5.7 Related Work

We focus our research on Model Fragment Location (MFL), so the most important knowledge areas where our research can be applied are: Bug Location, Feature Location, and Traceability Links Recovery. For this reason, in this section, we analyze some of the existing works in these areas and compare our work with them.

Most of the existing works focus on searches in source code; therefore, their measurements are oriented to measuring the number of code lines or the number of methods in the code. For instance, (Wong et al. 2016) present a systematic literature survey of bug location in source code. In (Dit et al. 2013),(Rubin and Chechik 2013), the authors present systematic literature surveys of feature location techniques in source code. Javed et al. (Javed and Zdun 2014) present a systematic literature review to discover the existing traceability approaches and tools between software architecture and source code. In contrast to them, our work focuses on models instead of source code, so the measurements that are presented in this work are oriented to measuring models (search space) and model fragments (solution space).

5.7.1 Related works on MFL on models

Some recent works center their efforts on MFL on models. Winkler et al. (Winkler and Pilgrim 2010) classify several approaches that have been created in the past 15 years which try to optimize automatic identification of traces in models. De Lucia et al. (De Lucia et al. 2004) present a Traceability Links Recovery method and tool, which are based on Latent Semantic Indexing (LSI) and include models. Spanoudakis et al. (Spanoudakis et al. 2004) present a linguistic rule-based approach to support the automatic generation of Traceability Links between requirements and models. In (Wille et al. 2013),(Holthusen et al. 2014),(Zhang, Øystein Haugen, and Møller-Pedersen 2011),(Zhang, O. Haugen, and Moller-Pedersen 2012),(Jabier Martinez et al. 2015) the authors focus on the location of features in models by comparing the models with each other to formalize the variability among them in the form of a Software Product Line. For instance, in (Holthusen et al. 2014), the authors present an improved version of a family mining approach for automatically discovering commonality and variability between related system variants. They apply their approach to function block diagrams that are used to develop automation software, and they show its feasibility in a manufacturing case study.

Wille et al. (Wille et al. 2013) present an approach for analyzing related models and determining the variability among them. Their analysis provides crucial information about the variability (changed parts, additional parts, and parts without any modification) between the models in order to create family models. In (Zhang, Øystein Haugen, and Møller-Pedersen 2011), an approach for synthesizing a software product line using model comparison is presented. During model difference detection, the approach applies EMF Compare, a generic model comparison tool. To specify the variability, the approach applies the Common Variability Language (CVL) (Øystein Haugen, Wasowski, and Czarnecki 2013), a generic language for expressing variability. Based on the comparison results, a preliminary product line model (CVL model) can be automatically induced and the SPL developer may further enhance the product line model. Just like us, the authors applied their research in the railway domain to illustrate their work.

Zhang et al. (Zhang, O. Haugen, and Moller-Pedersen 2012) present an approach for automating the augmentation of product lines using model comparison and variability modeling techniques. Their approach aims to reduce the manual effort involved in this process by automatically suggesting a tentative augmented product line model. The approach applies CVL Compare, a generic approach for automating the synthesis of a CVL-based product line from a set of existing product models. Martinez et al. (Jabier Martinez et al. 2015) introduced a generic and extensible framework for a bottom-up approach to Software Product Line Engineering. They presented the principles of the approach in order to reduce the current high up-front investment required for a systematic reuse end-to-end adoption.

All of these works (see the top of Table 5.4) are evaluated by means of different case studies that are measured to a greater or lesser extent. The most popular measurement reported is the size. However, none of them take into account the same measurements to work with models, and the measurements are selected by the researchers based on their own judgment. In contrast, we propose a set of measurements that are strongly related with models to measure the search space and the solution space.

5.7.2 Our previous related works on MFL on models

Finally, some of our previous works (J. Font et al. 2017), (Jaime Font et al. 2016), (Arcega, Jaime Font, Øystein Haugen, et al. 2016), (Marcén et al. 2017), (Lapeña Martí et al. 2017) present Feature Location approaches to discover software artifacts that implement the feature in models.

Marcen et al. (Marcén et al. 2017) propose a feature location approach to discover model elements that implement the feature functionality in a model. Through a model and a feature description, model fragments that are extracted from the model and the feature description are encoded based on a domain ontology. Then, a Learning-to-Rank algorithm is used to train a classifier that is based on the model fragments and encoded feature description. Lapeña et al. (Lapeña Martí et al. 2017) presented Computer Assisted Clone-and-Own form Models (CACAO4M), an approach for ranking relevant model fragments for the development of specific requirements for a new product. Through their approach, the authors aim to prioritize the model fragments that are easier to understand from the perspective of a software engineer.

Arcega et al. (Arcega, Jaime Font, Øystein Haugen, et al. 2016) propose an approach that combines architecture models at run-time and information retrieval for feature location. Specifically, their approach uses a scenario that executes the desired feature to be located. Also, the approach ranks all of

Table 5.4: Overview of related works regarding their searches and the five presented measurements: Size (S), Volume (V), Density(DE), Multiplicity (M), and Dispersion (DI)

	Related Works	Searches in Models	Measurements				
			Search Space		Solution		
			S	V	DE	M	DI
Other works	Wong et al. 2016	X	-	-	-	-	-
	Dit et al. 2013	X	-	-	-	-	-
	Rubin and Chechik 2013	X	-	-	-	-	-
	Javed and Zdun 2014	X	-	-	-	-	-
	Winkler and Pilgrim 2010	√	X	X	X	X	X
	De Lucia et al. 2004	√	X	X	X	X	X
	Spanoudakis et al. 2004	√	√	X	X	X	X
	Wille et al. 2013	√	√	X	X	X	X
	Holthusen et al. 2014	√	X	X	X	X	X
	Zhang, Øystein Haugen, and Møller-Pedersen 2011	√	√	X	X	X	X
	Zhang, O. Haugen, and Moller-Pedersen 2012	√	X	X	X	X	X
	Jabier Martinez et al. 2015	√	X	X	X	X	X
Our previous works	J. Font et al. 2017	√	√	√	X	X	X
	Jaime Font et al. 2016	√	√	√	X	X	X
	Arcega, Jaime Font, Øystein Haugen, et al. 2016	√	X	X	X	X	X
	Marcén et al. 2017	√	√	√	X	X	X
	Lapeña Martí et al. 2017	√	√	√	X	X	X

the model elements that are executed to extract the model elements that are related to the feature. Font et al. (Jaime Font et al. 2016) presented a Genetic Algorithm to Feature Location. They provide a custom encoding that enables the genetic algorithm to work with model fragments and a set of genetic operations that can be applied to individuals following that encoding. In addition, they present a fitness function, a parent selection operator, a crossover operation, and a mutation operation.

Font et al. (J. Font et al. 2017) propose and compare five search algorithms to locate features in a family of models: Evolutionary Algorithm (EA-MFL); Random Search (RS-MFL) used as a sanity check; steepest Hill Climbing (HC-MFL); Iterated Local Search with random restarts (ILS-MFL); and a hybrid between Evolutionary and Hill Climbing (EHC-MFL). They applied Latent Semantic Analysis (LSA) as the fitness function. Their results show that Search-based Software Engineering techniques can be applied to locate features in product models. They evaluate their work in two families of industrial models, demonstrating that Search-based Software Engineering for feature location at the model level can be applied in real-world environments.

Most of these works are evaluated by means of case studies that take into account the size of the models or the volume of models (see the bottom of Table 5.4). However, none of the case studies take into account all of the measurements that are presented in this paper: size, volume, density, multiplicity, and dispersion.

5.8 Conclusions

Traceability Links Recovery, Feature Location, and Bug Location are popular activities in the context of software maintenance. When the artifacts where the requirements, features, or bugs are located are models, many approaches focus on identifying relevant sets of model elements (Model Fragments). These Model Fragment Location (MFL) approaches leverage Information Retrieval, Linguistic techniques, and Search-Based Software Engineering techniques to locate the model fragments.

However, there is a lack of detail in the reporting of the measurements of both the search space and the solution, with model size being the only reported measure. Since different models are measured in different ways, model size values are not a valid comparison.

In this paper, we have proposed the usage of five measurements to report the results of MFL techniques. Apart from the size measurement, we introduced four novel measurements: volume, density, multiplicity, and dispersion. Of the five measurements, size and volume measure the search space, while density, multiplicity, and dispersion measure the solution.

In order to determine the relevance of the proposed measurements, we studied whether the values of the measurements have an impact on the results provided by two distinct MFL approaches. We evaluated these approaches in terms of precision and recall, and we analyzed their outcomes with regard to the measurements of the case study.

The results presented in the paper show that all of the proposed measurements have a direct impact on the results of the MFL approaches. Therefore, we strongly recommend their study and reporting. Furthermore, we alsion problems during MFL on industrial models. These values can be a reference for researchers who create synthetic location problems.

Bibliography

Arcega, Lorena, Jaime Font, Oystein Haugen, et al. (2017). “On the Influence of Models at Run-Time Traces in Dynamic Feature Location”. In: *Modelling Foundations and Applications*. Ed. by Anthony Anjorin and Huáscar Espinoza. Cham: Springer International Publishing, pp. 90–105. ISBN: 978-3-319-61482-3. DOI: 10.1007/978-3-319-61482-3_6 (cit. on p. 142).

Arcega, Lorena, Jaime Font, Øystein Haugen, et al. (2016). “Feature Location through the Combination of Run-Time Architecture Models and Information Retrieval”. In: *International Conference on System Analysis and Modeling*. Springer, pp. 180–195. ISBN: 978-3-319-46613-2 (cit. on pp. 164, 165).

Arcuri, Andrea and Lionel Briand (2014). “A Hitchhiker’s Guide to Statistical Tests for Assessing Randomized Algorithms in Software Engineering”. In: *Software Testing, Verification and Reliability* 24.3, pp. 219–250 (cit. on p. 159).

De Lucia, Andrea et al. (2004). “Enhancing an Artefact Management System with Traceability Recovery Features”. In: *Proceedings of the 20th IEEE*

International Conference on Software Maintenance. IEEE, pp. 306–315 (cit. on pp. 154, 163, 165).

Dit, Bogdan et al. (Jan. 2013). "Feature location in source code: A taxonomy and survey". In: *Journal of Software Maintenance and Evolution: Research and Practice* 25. DOI: `10.1002/smr.567` (cit. on pp. 162, 165).

Font, J. et al. (2017). "Achieving Feature Location in Families of Models through the use of Search-Based Software Engineering". In: *IEEE Transactions on Evolutionary Computation*, pp. 1–1. ISSN: 1089-778X. DOI: `10.1109/TEVC.2017.2751100` (cit. on pp. 142, 164–166).

Font, Jaime et al. (2016). "Feature Location in Model-Based Software Product Lines Through a Genetic Algorithm". In: *International Conference on Software Reuse*. Springer, pp. 39–54. ISBN: 978-3-319-35122-3. DOI: `10.1007/978-3-319-35122-3_3` (cit. on pp. 142, 164–166).

García, Salvador et al. (May 2010). "Advanced nonparametric tests for multiple comparisons in the design of experiments in computational intelligence and data mining: Experimental analysis of power". In: *Information Sciences* 180, pp. 2044–2064. DOI: `10.1016/j.ins.2009.12.010` (cit. on p. 159).

Haugen, Øystein, Andrzej Wasowski, and Krzysztof Czarnecki (Aug. 2013). "CVL: Common Variability Language". In: 2 (cit. on p. 163).

Holthusen, Sönke et al. (2014). "Family Model Mining for Function Block Diagrams in Automation Software". In: *18th International Software Product Lines Conference* (cit. on pp. 163, 165).

Javed, Muhammad Atif and Uwe Zdun (2014). "A systematic literature review of traceability approaches between software architecture and source code". In: *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. ACM, p. 16 (cit. on pp. 162, 165).

Koffka, Kurt (2013). *Principles of Gestalt Psychology*. Vol. 44. Routledge (cit. on p. 145).

Lapeña Martí, Raúl et al. (June 2017). "Model Fragment Reuse Driven by Requirements". In: (cit. on pp. 164, 165).

Marcén, Ana C. et al. (2017). “Towards Feature Location in Models Through a Learning to Rank Approach”. In: *Proceedings of the 21st International Systems and Software Product Line Conference - Volume B*. SPLC ’17. Sevilla, Spain: ACM, pp. 57–64. ISBN: 978-1-4503-5119-5. DOI: `10.1145/3109729.3109734` (cit. on pp. 164, 165).

Martinez, J. et al. (Nov. 2015). “Automating the Extraction of Model-Based Software Product Lines from Model Variants (T)”. In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 396–406. DOI: `10.1109/ASE.2015.44` (cit. on p. 142).

Martinez, Jabier et al. (2015). “Bottom-up Adoption of Software Product Lines: a Generic and Extensible Approach”. In: *Proceedings of the 19th International Conference on Software Product Lines* (cit. on pp. 163–165).

Rubin, Julia and Marsha Chechik (2013). “A Survey of Feature Location Techniques”. In: *Domain Engineering*. Springer, pp. 29–58 (cit. on pp. 162, 165).

Spanoudakis, George et al. (2004). “Rule-Based Generation of Requirements Traceability Relations”. In: *Journal of Systems and Software* 72.2, pp. 105–127 (cit. on pp. 154, 163, 165).

Störrle, Harald (2014). “On the Impact of Layout Quality to Understanding UML Diagrams: Size Matters”. In: *MODELS* (cit. on pp. 145, 146).

Wille, David et al. (2013). “Interface Variability in Family Model Mining”. In: *17th International Software Product Line Conference* (cit. on pp. 163, 165).

Winkler, Stefan and Jens Pilgrim (2010). “A Survey of Traceability in Requirements Engineering and Model-Driven Development”. In: *Software and Systems Modeling (SoSyM)* 9.4, pp. 529–565 (cit. on pp. 142, 154, 163, 165).

Wong, W Eric et al. (2016). “A Survey on Software Fault Localization”. In: *IEEE Transactions on Software Engineering* 42.8, pp. 707–740 (cit. on pp. 162, 165).

Zhang, Xiaorui, O. Haugen, and B. Moller-Pedersen (Dec. 2012). “Augmenting Product Lines”. In: *Software Engineering Conference (APSEC), 2012 19th Asia-Pacific*. Vol. 1, pp. 766–771. DOI: `10.1109/APSEC.2012.76` (cit. on pp. 163, 165).

Zhang, Xiaorui, Øystein Haugen, and Birger Møller-Pedersen (2011). “Model Comparison to Synthesize a Model-Driven Software Product Line”. In: *Proceedings of the 15th International Conference on Software Product Lines* (cit. on pp. 163, 165).

Chapter 6

On the influence of Model Fragment Properties in Machine Learning-based Feature Location

Context: *Leveraging machine learning techniques to address feature location on models has gained attention in the Software Product Line community. Machine learning techniques empower the software product companies to take advantage of the knowledge and the experience generated for years in their companies to improve the performance of the feature location process. Nonetheless, in feature location on models, the knowledge base contains different artifacts from the ones used in feature location on source code. In fact, the model fragments in the knowledge bases, which are used for feature location on models, have properties that may influence results.*

Objective: *In this paper, we analyze the influence of three model fragment properties in feature location: density, multiplicity, and dispersion.*

Method: *The analysis of these properties was based in an industrial case study (268 test cases and 7500 samples in the knowledge base) provided by CAF, a worldwide provider of railway solutions.*

Results: *The density and dispersion properties have a direct impact on the feature location results. The model fragments with extra-small density values achieve results with up to 43% more precision, 41% more recall, 42% more F-measure, and 0.53 more Matthews Correlation Coefficient (MCC) than the model fragments with other density values. On the other hand, the model fragments with extra-small and small dispersion values achieve results with up to 53% more precision, 52% more recall, 52% more F-measure, and 0.57 more MCC than the model fragments with other dispersion values.*

Conclusions: *The statistical analysis of the results shows that both density and dispersion properties significantly influence the results in our case study. Therefore, the results of this work can serve not only to improve the reporting of machine learning-based feature location on models, but also to fairly compare approaches and thereby improve the feature location results.*

6.1 Introduction

Feature location is a key activity in reengineering a set of products in a Product Line. Feature location is known as the process of finding the set of software artifacts that realize a specific feature (Font et al. 2017). In the case of feature location in models, the goal is to identify the model fragment that is associated to a specific feature. Therefore, a model fragment contains the elements of the product model that make the software functionality described in a feature possible.

Increasingly, some recent works have focused on machine learning techniques to address the challenge of feature location (Corley, Damevski, and Kraft 2015),(B. Le et al. 2016),(Ye, Bunescu, and C. Liu 2014),(Ana C. Marcén et al. 2017),(Binkley and Lawrie 2014). In these works, machine learning techniques take advantage of the knowledge and the experience generated for years in software companies to perform the feature location process. Specifically, the knowledge and experience make up part of the knowledge bases, which are used by machine learning techniques to learn how to locate features. To report machine learning-based feature location problems, most of the works describe in detail the machine learning techniques, the tuning parameters, and the knowledge bases. Nonetheless, in feature location on models, the knowledge bases contain different artifacts from the ones used in feature location on source code.

Figure 6.1 shows an example of a sample of a knowledge base for feature location in models. Each sample in the knowledge base is composed of a feature description, a model fragment, and a score. In the sample of this example, the feature, which is described using natural language, is related to a model fragment by means of a score. The model fragment, which is highlighted in gray, consists of a set of three model elements (*Pantograph 2*, *Circuit Breaker 2*, and the relation between them). These three elements belong to a model that has more elements that are not included in the model fragment, such as *Pantograph 1* or *Converter 1*. The score determines the similarity between the feature description and the model fragment. In this example, a score equal to 4 means that the model fragment contains all of the model elements associated to a feature. In contrast, a score equal to 0 means that none of the elements of the model fragments are related to the feature.

In this paper, we analyze the influence of three model fragment properties (Ballarin et al. 2018) density, multiplicity, and dispersion. The analysis of these properties is based on an industrial case study (268 test cases and 7500

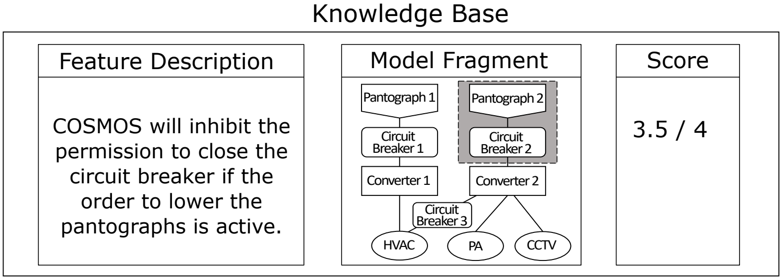


Figure 6.1: Example of a sample of the knowledge base for feature location on models

samples in the knowledge base) provided by CAF[1], a worldwide provider of railway solutions. The test cases were evaluated taking into account different values of the properties of the model fragments in the knowledge base. For example, one of the performed tests consisted of the evaluation of the test cases using only the model fragments in the knowledge base with small density values.

Our results show that the multiplicity property does not have an impact on the feature location results. However, the density and dispersion properties do have a direct impact on the feature location results. In our case study, the model fragments with extra-small density values achieved results of up to 43% more precision, 41% more recall, 42% more F-measure, and 0.53 more MCC than the model fragments with other density values. On the other hand, the model fragments with extra-small and small dispersion values achieved results of up to 53% more precision, 52% more recall, 52% more F-measure, and 0.57 more MCC than the model fragments with other dispersion values. The statistical analysis of the results show that both density and dispersion properties significantly influenced the results. Moreover, this analysis shows the magnitude of the influence of the density and dispersion in our case study.

Taking into account the results of the evaluation, research works should report density and dispersion in order to enable replication of their works. Moreover, these results also provide evidence that model fragment properties are relevant in order to be able to fairly compare properties and improve the feature location results. If we wanted to fairly compare two machine learning-based approaches for feature location in models, we would have to take into account that both approaches can need model fragments with different property values for their knowledge bases. For example, a deep learning technique usually requires a knowledge base with a greater number of samples than a learning-

[1] www.caf.net/en

to-rank algorithm, although both are machine learning techniques. Therefore, an approach based on deep learning requires a knowledge base with a different number of samples than an approach that is based on learning to rank. Likewise, an approach based on deep learning may need model fragments with different density values for the knowledge base than an approach based on learning to rank. Therefore, in order to obtain the best performance of the compared approaches, the model fragment properties for the knowledge base must be configured for each approach.

On the other hand, the evaluation results show that by taking into account specific values for one property, we can improve the feature location results. For example, by taking into account only extra-small density values, we can surpass the results obtained using other density values. Likewise, by taking into account extra-small and small dispersion values, we can surpass the results obtained using other dispersion values. Therefore, we can improve the feature location results of our case study by configuring the knowledge base using model fragments with extra-small density or extra-small and small dispersion. In summary, in machine learning-based feature location on models, the results of this work can serve not only to improve the reports by means of the model fragment properties, but can also fairly compare machine learning-based feature location approaches and thereby improve the feature location results.

The remainder of this paper is structured as follows: Section 6.2 provides background on our case study. Section 6.3 highlights the properties of knowledge bases that include the model fragment properties. Section 6.4 details the means used to evaluate our work, the results of the evaluation, and the statistical significance of the obtained results. Section 6.5 discusses the performed experiment and the obtained results. Section 6.6 describes the threats to the validity of our work. Section 6.7 introduces the existing works that are related to our work. Finally, Section 6.8 concludes the paper.

6.2 Background

This section presents the Train Control and Management Language (TCML), which is used to formalize the products manufactured by our industrial partner. Then, an illustration of an equipment-focused simplified subset of the DSL is presented. In addition, we present the Common Variability Language (CVL) (Haugen et al. 2008), which is the language used to formalize the model fragments.

The Train Control and Management Language (TCML) has the expressiveness required to describe both the iteration between the main pieces of equipment installed in a train unit and the non-functional aspects related to regulation. A train unit is furnished with multiple pieces of equipment. Some examples are: the traction equipment, the compressors that feed the brakes, the pantograph that harvests power from the overhead wires, and the circuit breaker that isolates or connects the electrical circuits of the train. The TCML will be used through the rest of the paper to present a running example.

For the sake of understandability and legibility, and due to intellectual property rights concerns, we present a simplified equipment-focused subset of the DSL. The top of Figure 6.2 depicts one example, which is taken from a real-world train. It presents a converter assistance scenario where two separate pantographs (High Voltage Equipment) collect energy from the overhead wires and send it to their respective circuit breakers (Contactors), which in turn send it to their independent Voltage Converters. The converters then power their assigned Consumer Equipment: the HVAC on the left (the train's air conditioning system), and the PA (public address system) and CCTV (television system) on the right.

To formalize the Model fragments used throughout the rest of our work, we use the Common Variability Language (CVL) (Haugen et al. 2008), which expresses variability among models in terms of Model Fragments such as Placement Fragments (variation points) and Replacement Fragments (variants). The materialization of product models is performed by means of Fragment Substitutions between a Base Model (Placements) and a Model Library (Replacements). In the top right of Figure 6.2, the element highlighted in gray is an example of a model fragment, which includes one circuit breaker that connects Converter 2 to a Consumer Equipment assigned to Converter 1. This model fragment is the realization of the "converter assistance" feature, which allows the passing of current from one converter to equipment assigned to its peer for coverage in case of overload or failure of the first converter. A simple example of model fragment manipulation in the Train Control and Management Language (TCML) can be found at: youtube.com/watch?v=Ypcl2evEQB8

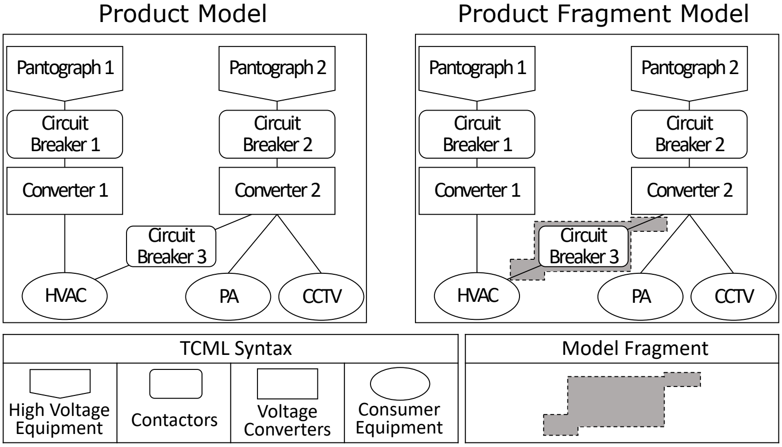


Figure 6.2: Example of a TCML model, model fragment, and sample of the knowledge base

6.3 Properties of the knowledge base

The knowledge base condenses the knowledge and the experience that have been generated in the companies for years in order to take advantage of this information. In machine learning-based feature location, the knowledge bases are used to learn how to automatically locate features. To do this, the known features stored in the knowledge base are compared by means of different machine learning techniques to find similarities and differences that can help to locate unknown features. Specifically, in feature location in models, the knowledge base is composed of samples, where each sample contains a feature description, a model fragment, and a score. Figure 6.1 shows an example of a sample of a knowledge base for feature location in models.

Since the success of the feature location depends on properly learning how to locate unknown features, the knowledge base must contain the information necessary to learn how to do it. In other words, the completeness and heterogeneity of the knowledge base facilitates or limits the learning of how to locate unknown features. For this reason, both the size and the class distribution of the knowledge base have been widely discussed in the literature:

- **Size** is the number of samples in the knowledge base, which can greatly impact classification accuracy. As a consequence, the analysis of the size has been a major point of interest in research, generally finding a positive relationship between the size of the knowledge base and classification

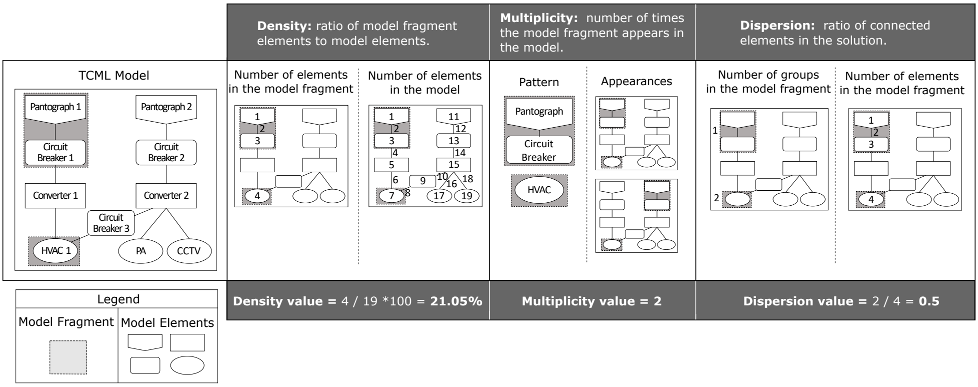

Density: ratio of model fragment elements to model elements.
Multiplicity: number of times the model fragment appears in the model.
Dispersion: ratio of connected elements in the solution.
TCML Model
Pantograph 1
Pantograph 2
Circuit Breaker 1
Circuit Breaker 2
Converter 1
Converter 2
Circuit Breaker 3
HVAC 1
PA
CCTV
Number of elements in the model fragment
Number of elements in the model
Pattern
Appearances
Pantograph
Circuit Breaker
HVAC
Number of groups in the model fragment
Number of elements in the model fragment
Density value = 4 / 19 *100 = 21.05%
Multiplicity value = 2
Dispersion value = 2 / 4 = 0.5
Legend
Model Fragment
Model Elements

Figure 6.3: Example of a model fragment with its values for the properties: density, multiplicity, and dispersion

accuracy for a wide range of classifiers (Zhuang et al. 1994),(Foody and Mathur 2004),(Foody, Mathur, et al. 2006).

- **Class distribution** is the percentage of samples for each possible class in the knowledge base. It is conventional wisdom that classifiers tend to perform worse when the knowledge base is imbalanced. For this reason, many research works have been tackling the problems of learning from data sets with imbalanced distributions and where the costs of misclassifying examples is non-uniform (Weiss and Provost 2001),(Weiss and Provost 2003),(Buda, Maki, and Mazurowski 2018). In our case, as the example of Figure 6.2 shows, the samples of our knowledge base depend on their scores. Therefore, our knowledge base depends on a score distribution that is balanced when there is the same number of samples with scores between 0 and 1, between 1 and 2, between 2 and 3, and between 3 and 4.

We consider that these properties should be used to report our knowledge bases. Nevertheless, we also think that the nature of the problem may require other more specific properties in order to provide all of the necessary information for the replication of the experiments. For example, the properties identified in (Ballarin et al. 2018) to report the search space and the solutions in feature location problems are specifically based on models. In fact, of the identified properties, three of them (density, multiplicity, and dispersion) focus on reporting property model fragments.

- **Density** measures the percentage of model elements that are present in a model fragment. The density is computed as the ratio of model fragment elements to model elements. Figure 6.3 shows an example of how to calculate the density of a model fragment. In this example, the model has 19 elements in total. The model fragment contains four elements (the *Pantograph 1*; the relation between *Pantograph 1* and *Circuit Breaker 1*; the *Circuit Breaker 1*; and the *HVAC*). Therefore, the density value is equal to 21.05%, which means that the model fragment is composed of 21.05% of the model elements.

- **Multiplicity** measures the number of times that the model fragment appears in the model. Figure 6.3 shows an example of how to calculate the multiplicity of a model fragment. In this example, the model fragment contains four elements: a pantograph that is connected to a circuit breaker; the relation to connect the pantograph and the circuit breaker; the circuit breaker connected to the pantograph; and a HVAC. Taking into account these elements, the multiplicity value is equal to 2, which means that there are two possible model fragments that contain these elements with their connections.

- **Dispersion** measures the ratio of connected elements in the model fragment. Specifically, a model fragment is composed of the model elements, but these elements may or may not be connected in the model. Therefore, the values for this property are from 0 to 1, where values around 0 indicate a strong connection among the elements of the model fragment and values around 1 indicate a strong dispersion among the elements of the model fragment. Figure 6.3 shows an example of how to calculate the dispersion of a model fragment. In this example, the model fragment contains four elements: the *Pantograph 1*, the relation between *Pantograph 1* and *Circuit Breaker 1*, the *Circuit Breaker 1*, and the *HVAC*. The first three elements are connected so they compose the first group; since the last element is not connected to the other elements, it composes the second group. Since the dispersion is computed as the ratio between the number of groups and the number of elements, the dispersion value is equal to 0.5, which means that the model fragment has a medium dispersion.

Based on these properties, three domain experts discussed whether any of these three properties should be used to populate the knowledge base. The first expert believed that none of the properties would have an impact on the results. The second one believed that since density and dispersion would have an impact on the results, the knowledge base should be balanced with regard to these properties. The third one believed that since the three properties would have an impact on the results, the knowledge base should be balanced with regard the three properties. However, the greater the number of properties that have to be balanced, the more expensive and complex the generation of the knowledge base will be. For example, consider a knowledge base where half of the samples have a score value equal to *0* and half of the samples have a score value equal to *1*. If the knowledge base must be balanced from both the score values (*0* or *1*) and the density values (*lesser than <=50* or *>50*), the knowledge base must have the same number of samples for all of the possible combinations between the score values and the density values: samples with a score value equal to *0* and a density value *<=50*; samples with a score value equal to *0* and a density value *>50*; samples with a score value equal to *1* and a density value *<=50*; and samples with a score value equal to *1* and a density value *>50*. In this example, there are only four combinations, but the problem appears when the properties have more than two values and there are more than two properties to be balanced.

In fact, our industrial partner has been developing firmware since 1995 and 7,500 samples have been documented in their knowledge base. However, if the knowledge base had to be balanced taking into account only one of the properties without losing the balanced score distribution, the knowledge base would be approximately reduced to 300 samples. Also the balance regarding all of the properties would reduce the knowledge base to a few hundred samples.

Therefore, event though the different beliefs of the domain experts indicate that density, multiplicity, or dispersion in the model fragments of the knowledge base may have an impact on the results, the problem is complex enough to require an experiment in order to reach an agreement. Since balancing a knowledge base according to the density, multiplicity, and dispersion values would require a high cost, before balancing the knowledge base, an experiment would help determine whether or not balancing any of the properties makes sense. These two facts have motivated us to propose the research questions and the experiment that are defined in Section 6.4.

6.4 Evaluation

Since the model fragments of the knowledge base have different properties (density, multiplicity, and dispersion), the goal of this paper is to research the influence of these properties on the results in model fragment location. To do this, this paper provides answers to several research questions. The first three research questions are focused on determining which property influences the results:

RQ1: Do the density values of the model fragments in the knowledge base influence the results?

RQ2: Do the multiplicity values of the model fragments in the knowledge base influence the results?

RQ3: Do the dispersion values of the model fragments in the knowledge base influence the results?

The following two research questions are focused on determining how the properties influence the results. Therefore, taking into account the previous research questions, this paper provides answers to the following questions for each property that influences the results:

RQ4: What values of a property should be covered by the model fragments in the knowledge base to obtain the best results?

RQ5: If the solution searched for in the test cases have different property values than the model fragments in the knowledge base, is it possible to obtain good results?

This section presents the evaluation that was performed to answer the RQs. It includes a description of the experimental setup, a description of the case study, the implementation details, the results obtained, and the statistical analysis performed.

6.4.1 Experimental Setup

The goal of this experiment is to provide answers to the RQs. To do this, the experiment was addressed by means of the Feature Location in Models based on the Machine Learning (FLiM-ML) approach. However, instead of using the entire knowledge base, we performed several selections of the knowledge base covering different values of each property. Then, the approach was used to evaluate all of the test cases taking into account each selection. Finally, the

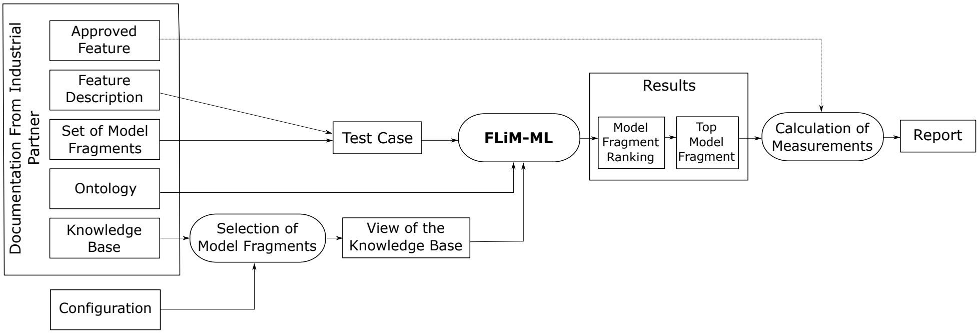


Figure 6.4: Experimental Setup

results of the approach were analyzed to provide the required answers. Figure 6.4 shows an overview of the process.

The left part of Figure 6.4 shows the inputs. Most of these inputs are extracted from the documentation provided by our industrial partner. The first input that is provided by our industrial partner is the **approved solution**. The approved solution is the best realization of a feature. We know beforehand what the best realization for each described feature is. This helps us to compare the results of the approach with the expected solution in order to determine how good the obtained results are.

The following two inputs are the **feature descriptions** and the **sets of model fragments**. Each feature description defines in natural language a feature that is searched for in a set of model fragments. Each set of model fragments is composed of different model fragments that include the realization of the searched feature. Specifically, each feature description and the set of model fragments, where this feature is going to be searched, make up a test case. All of the test cases are evaluated through the FLiM-ML approach.

On the other hand, the **ontology** and the **knowledge base** are necessary so that the approach can be used to evaluate the test cases. The ontology contains the main concepts, properties, and relations of a domain. Specifically, the ontology is used by the approach to characterize and encode the model fragments and feature descriptions into a format that is understandable for the machine learning techniques. In contrast, the knowledge base is used by the approach to learn how to evaluate the test cases. The knowledge base is composed of samples, where each sample relates a feature description and a model fragment according to a score. Specifically, since the model fragment realizes the feature description to a greater or lesser extent, this score indicates the degree of similarity between the model fragment and feature description.

The last input is the set of **configurations** that are defined to answer the RQs. Specifically, from the same knowledge base, each configuration indicates what samples are going to be selected to evaluate the all of the test cases. Therefore, these configurations are defined to cover different values of the property that is being analyzed. In order to answer RQ1, we defined five configurations in order to select samples with different density values:

- **Density-XS** selects samples that have a density value between 0 and 25. Therefore, the view of the knowledge base that is generated by means of this configuration only contained samples whose model fragment had an extra-small density value.
- **Density-S** selects samples that have a density value between 25 and 50. Therefore, the view of the knowledge base that was generated by means of this configuration only contained samples whose model fragment had a small density value.
- **Density-M** selects samples that have a density value between 50 and 75. Therefore, the view of the knowledge base that was generated by means of this configuration only contained samples whose model fragment had a medium density value.
- **Density-L** selects samples that have a density value between 75 and 100. Therefore, the view of the knowledge base that was generated by means of this configuration only contained samples whose model fragment had a large density value.
- **Density-A** selects samples that have a density value between 0 and 100. Specifically, a quarter of the selected samples had a density value between 0 and 25, a quarter of the selected samples had a density value between 25 and 50, a quarter of the selected samples had a density value between 50 and 75, and a quarter of the selected samples had a density value between 75 and 100. Therefore, the view of the knowledge base that was generated by means of this configuration contained samples covering all density values.

In order to answer RQ2, we defined three configurations to select samples with different multiplicity values:

- **Multiplicity=1** selects samples that have a multiplicity value equal to 1. Therefore, the view of the knowledge base that was generated by means of this configuration only contained samples whose model fragment appeared one time in the product model.
- **Multiplicity>1** selects samples that have a multiplicity value greater than 1. Therefore, the view of the knowledge base that was generated by means of this configuration only contained samples whose model fragment appeared two or more times in the product model.
- **Multiplicity-A** selects samples that have any multiplicity value. Specifically, half of the selected samples had a multiplicity value equal to 1, and half of the selected samples had a multiplicity value greater than 1. Therefore, the view of the knowledge base that was generated by means of this configuration contained samples covering all multiplicity values.

In order to answer RQ3, we defined three configurations to select samples with different dispersion values:

- **Dispersion-XS** selects samples that have a dispersion value between 0 and 0.25. Therefore, the view of the knowledge base that was generated by means of this configuration only contained samples whose model fragment had an extra-small dispersion value.
- **Dispersion-S** selects samples that have a dispersion value between 0.25 and 0.50. Therefore, the view of the knowledge base that was generated by means of this configuration only contained samples whose model fragment had a small dispersion value.
- **Dispersion-M**, selects samples that have a dispersion value between 0.50 and 0.75. Therefore, the view of the knowledge base that was generated by means of this configuration only contained samples whose model fragment had a medium dispersion value.
- **Dispersion-L** selects samples that have a dispersion value between 0.75 and 1. Therefore, the view of the knowledge base that was generated by means of this configuration only contained samples whose model fragment had a large dispersion value.
- **Dispersion-A** selects samples that have a dispersion value between 0 and 1. Specifically, a quarter of the selected samples had a dispersion value between 0 and 0.25, a quarter of the selected samples had a dispersion value between 0.25 and 0.50, a quarter of the selected samples had a

dispersion value between 0.50 and 0.75, and a quarter of the selected samples had a dispersion value between 0.75 and 1. Therefore, the view of the knowledge base that was generated by means of this configuration contained samples covering all dispersion values.

Each configuration is used to generate a different view of the knowledge base, and each view is used to evaluate all of the test cases. Each combination (between a view of the knowledge base and a test case) provides a ranking of model fragments as output. The top fragment of this ranking is the best realization of the feature found by means of the FLiM-ML approach. Therefore, this top model fragment is compared with the correspondent approved feature in order to determine how good the obtained model fragment is. This comparison is performed by means of a confusion matrix.

A confusion matrix is a table that is often used to describe the performance of a classification model (in this case, the performance of each configuration) on a set of test data (the test cases) for which the true values are known (from the approved features). In our case, each model fragment in the test cases obtains a score in the feature location process. Since the granularity is at the level of model elements, the presence or absence of each model element is considered to be a classification. The confusion matrix distinguishes between the predicted values and the real values, classifying them into four categories:

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

Then, some performance measurements are derived from the values in the confusion matrix. Specifically, we create a report that includes four performance measurements (recall, precision, the F-measure, and the MCC) for each combination of a configuration and a test case.

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

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

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

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

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

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

However, none of these previous measures correctly handle negative examples (TN). The **MCC** is a correlation coefficient between the observed and predicted binary classifications that takes into account all of the observed values (TP, TN, FP, FN) and is defined as follows:

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

Recall values can range between 0% (i.e., no single model element from the oracle is present in the model fragment of the solution) and 100% (i.e., all of the model elements from the oracle are present in the solution). Precision values can range between 0% (i.e., no single model element from the solution is present in the oracle) and 100% (i.e., all of the model elements from the solution are present in the oracle). A value of 100% precision and 100% recall means that both the solution and the feature realization from the oracle are the same. MCC values can range between −1 (i.e., there is no correlation between the predicted solution and the oracle) to 1 (i.e., the predicted solution is perfect). Moreover, a MCC value of 0 corresponds to a random prediction.

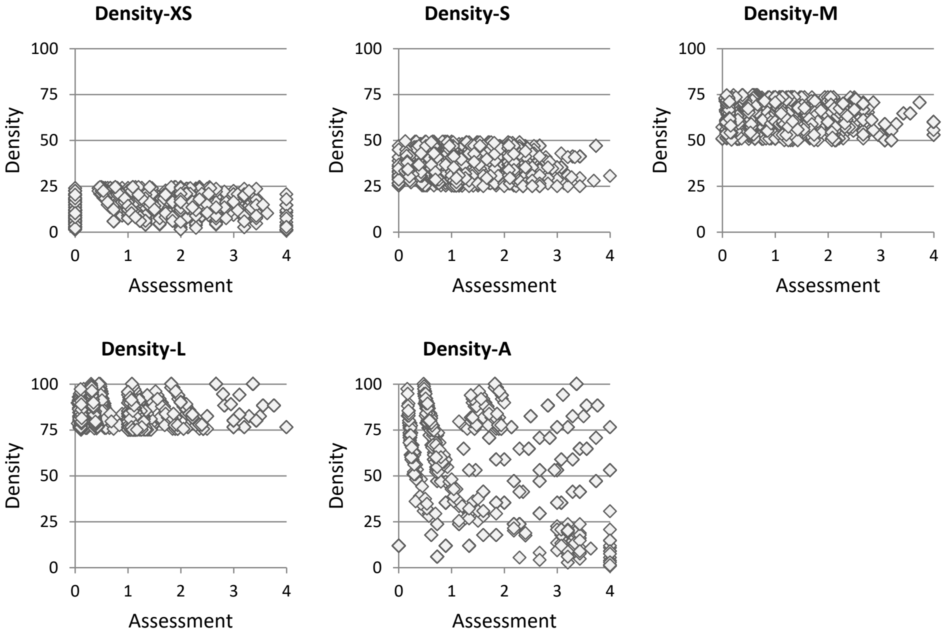
6.4.2 Case Study

The case study where we applied feature location was CAF, a worldwide provider of railway solutions. Their trains can be found 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 in its vehicles and cabins. These pieces of equipment are often designed and manufactured by different providers, and their aim is to carry out specific tasks for the train. Some examples of these devices are: the traction equipment, the compressors that feed the brakes, the pantograph that harvests power from the overhead wires, and 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 of the equipment cooperate in order to achieve the train functionality, while guaranteeing compliance with the specific regulations of each country. The following video illustrates the CAF models: `www.youtube.com/watch?v=Ypcl2evEQB8`.

Our evaluation includes 268 test cases and 13 configurations of the knowledge base that are defined in the experimental setup. Each test case is composed of a feature description and a set of model fragments. Specifically, the feature description is defined using natural language and contains about 25 words. The set of model fragments has about 100 model fragments with different values of density, multiplicity, and dispersion.

Moreover, our case study also includes the ontology and the knowledge base that are necessary for the feature location approach. The ontology contains a total of 54 elements between concepts and relations. The knowledge base has about 7500 samples. These samples were selected according to each configuration generating 13 different views of the knowledge base, one for each configuration. Specifically, each view has about 1600 samples, where about 400 samples have a score between 0 and 1, about 400 samples have a score between 1 and 2, about 400 samples have a score between 2 and 3, and about 400 samples have a score between 3 and 4. Therefore, score values can range between 0 (i.e., no single model element from the oracle is present in the solution) and 4 (i.e., all of the model elements from the oracle are present in the solution). Moreover, the density, multiplicity, and dispersion values for each view depend on the configuration used. The first five configurations, which are related to RQ1, allow us to generate five views of the knowledge base based on density. Figure 6.5 shows the distribution of these views according to the score and the density value of the samples selected for these views.

Then, the following three configurations, which are related to RQ2, allow us to generate three views of the knowledge base based on multiplicity. Figure 6.6


Density-XS

Density-S

Density-M

Density-L

Density-A

Figure 6.5: Distribution of the model fragments selected from the first five configurations (Density-XS, Density-S, Density-M, Density-L, and Density-A) according to their scores and density values

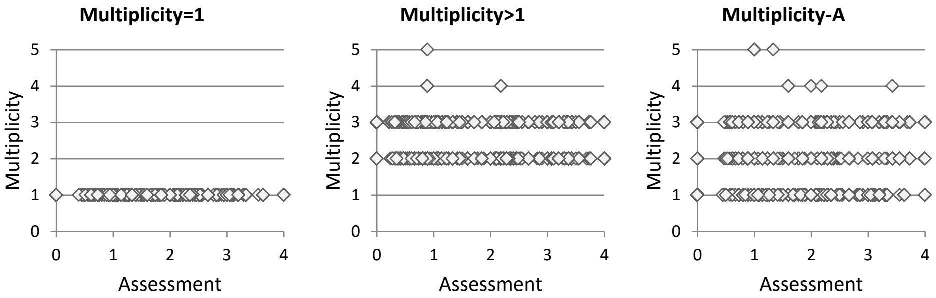


Figure 6.6: Distribution of the model fragments selected from the following three configurations (Multiplicity=1, Multiplicity>1, Multiplicity-A) according to their scores and multiplicity values

shows the distribution of these views according to the score and the multiplicity value of the samples selected for these views.

Finally, the last five configurations, which are related to RQ3, allow us to generate five views of the knowledge base based on dispersion. Figure 6.7 shows the distribution of these views according to the score and the dispersion value of the samples selected for these views.

For each view of the knowledge base, we evaluated the all of the test cases. Moreover, each combination of view and test case was run 30 times. As suggested by (Arcuri and Fraser 2013), given the stochastic nature of the evaluation, several repetitions are needed to obtain reliable results. Finally, the results were compared to the approved features (oracle). In our case study, the approved features consist of a set of model fragments, where each model fragment contains the model elements that are required by a feature description. In other words, the approved features contain the solutions for each test case, so the oracle had 268 model fragments (one for each test case). Figure 6.8 shows the distribution of the oracle regarding the density, multiplicity, and dispersion properties. Moreover, since the model fragments in the oracle are the correct solutions, they always have a score equal to 4. Therefore, the score of the model fragments in the oracle is not considered in the plots.

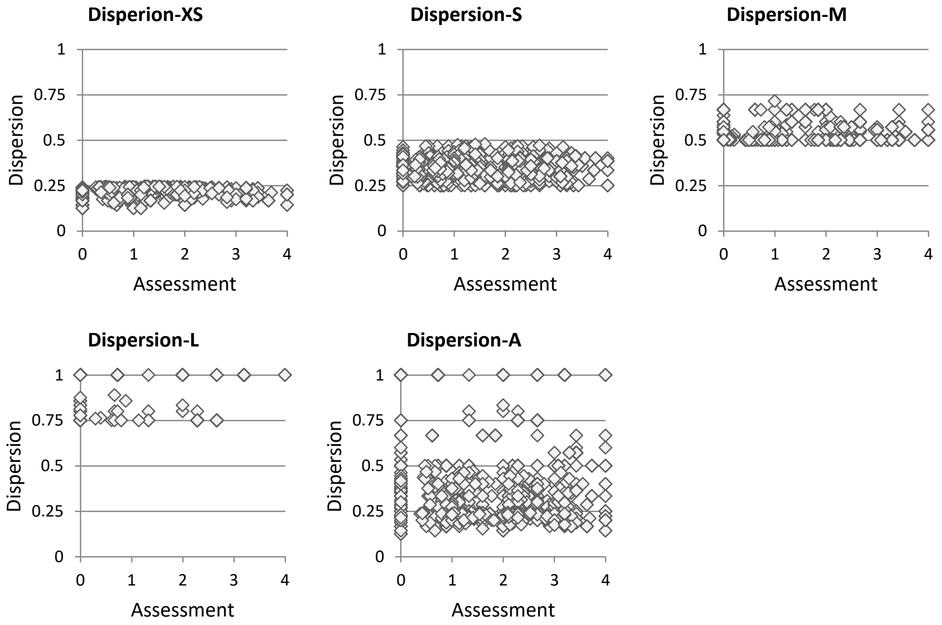


Figure 6.7: Distribution of the model fragments selected from the last five configurations (Dispersion-XS, Dispersion-S, Dispersion-M, Dispersion-L, and Dispersion-A) according to their scores and dispersion values

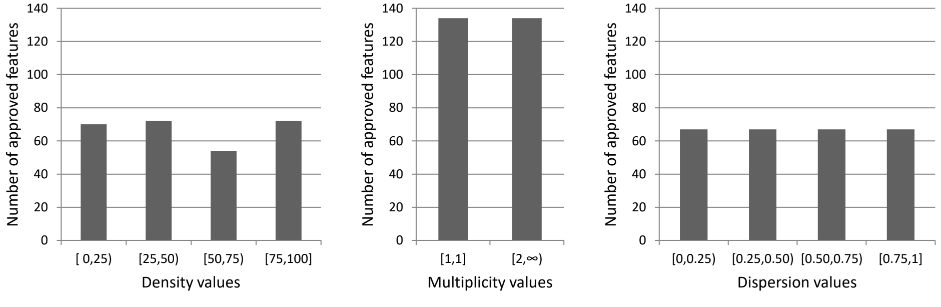


Figure 6.8: Distribution of the oracle regarding the number of approved features for the different density, multiplicity, and dispersion values

6.4.3 FLiM-ML Approach

The FLiM-ML (Ana C. Marcén et al. 2017) is an approach for locating features in models that are based on machine learning. In this work, the FLiM-ML approach was used to evaluate the different configurations that serve to answer the RQs of this work. Figure 6.9 shows an overview of the FLiM-ML approach, whose objective is to provide a ranking of model fragments. The top model fragment in the ranking is the best realization found for a feature description. To do this, the approach has two phases: training and testing.

In the training phase, a classifier is trained to learn how well each model fragment realizes a specific feature description. To do this, the input consists of an ontology and a knowledge base. The ontology is composed of concepts, properties, and relations of a domain. The knowledge base is composed of samples, where each sample relates a feature description and a model fragment according to a score. Specifically, since the model fragment realizes the feature description to a greater or lesser extent, this score indicates the degree of similarity between the model fragment and the feature description. In our case, instead of using the whole knowledge base, we used the views of the knowledge base that are described in the experimental setup and the case study.

The training phase consists of four steps:

1. Encoding: the ontology is used to encode the samples of the knowledge base into feature vectors, as described in (Ana C Marcén, Pérez, and Cetina 2017). Moreover, since both feature descriptions and model fragments are based on natural language, the terms used in the ontology do not always align well with the terms in the feature description and with the terms in the model fragments. For this reason, before encoding, the feature descriptions and the model fragments are processed by a combination of NLP techniques defined in (Lapeña et al. 2017), which consists of tokenizing, lowercasing, removal of duplicate keywords, syntactical analysis, lemmatization, and stopword removal.

2. Feature selection: applies a mask to select only the most relevant features in the feature vectors. To do this, a set of masks is generated and evolved by means of an evolutionary algorithm as in (Ana C Marcén, Pérez, and Cetina 2017). As a result, the top mask in the ranking, which is obtained from the evolutionary algorithm, indicates the selected characteristics.

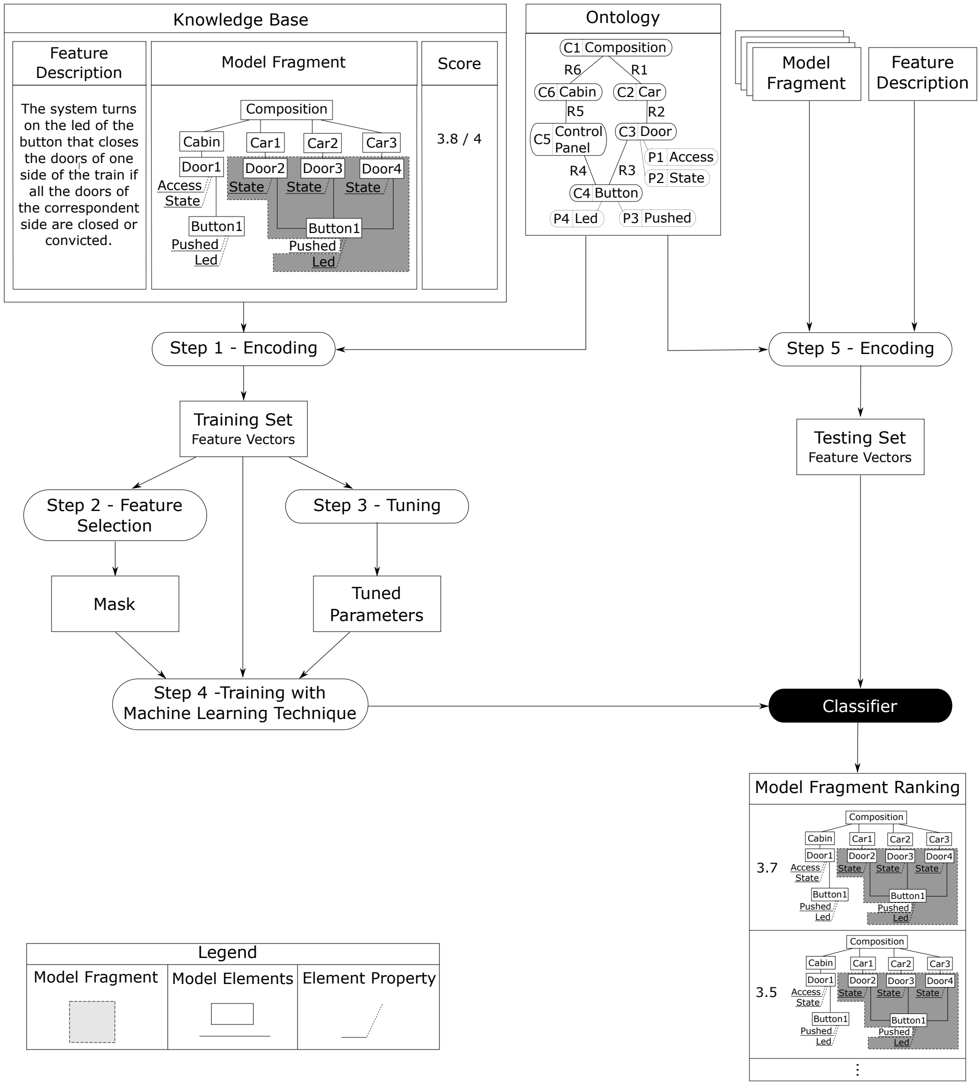


Figure 6.9: Overview of the FLiM-ML approach

3. Tuning: determines what parameters must be used to obtain the best performance of the machine learning technique. The parameter tuning in the FLiM-ML approach is performed in (Kıraç, Aktemur, and Sözer 2018). First, a grid search is built to determine the values of the parameters, which depend on the machine learning technique selected. Then, the FLiM-ML approach uniformly samples each of the parameters in their range and evaluates all of the combinations of the sampled values.

4. Training with a machine learning technique: the training set is used to train the classifier, which learns a rule-set through the comparison of the feature vectors of the training set (Shabtai et al. 2009). However, before using this classifier in the testing phase, it is worth analyzing the performance of the classifier through cross-validation. Cross-validation is a statistical method of evaluating and comparing machine learning techniques by dividing data into two segments: one used to train a classifier, and the other used to validate the classifier (Refaeilzadeh, Tang, and H. Liu 2009). Moreover, to reduce variability, multiple rounds of cross-validation are performed using different partitions, and the results are averaged over the rounds (Song et al. 2011).

   The results of the cross-validation provide the performance of the classifier. If this performance is not considered suitable, it is necessary to perform another training iteration. In this iteration, some artifacts of the training phase (e.g., the encoding, the ontology, the knowledge base, or the machine learning technique) must be modified in order to improve the classifier. Otherwise, if the performance is considered suitable, the classifier obtains the go-ahead, so the classifier trained with the whole knowledge base is used in the testing phase. Once the classifier has been generated, the training phase does not have to be repeated again. The same classifier is used to evaluate all of the test cases for a configuration in the testing phase. Therefore, the classifier is considered to be both an artifact (output of Step 4 in the training phase) and a step (responsible for testing the test cases in the testing phase). For this reason, Figure 6.9 shows the classifier in a black, rounded rectangle to point out its double meaning.

In the testing phase, the classifier is used to rank the set of model fragments according to a feature description described in natural language. To do this, the input consists of the set of model fragments, the feature description, and an ontology. Each model fragment realizes the feature description to a greater or lesser extent.

The testing phase consists of two steps:

5. Encoding: The ontology is used to encode each model fragment and the feature description. To be fair, both the characteristics of the encoding and the ontology must be the same for the the training phase and the testing phase. As a result, each model fragment and the feature description are encoded as a feature vector in the testing set.

6. Classifier: Then, each feature vector of the testing set is tested by the classifier, which uses the learned rule-set to assign a score to each one of them. This score is a numerical value that is greater than 0. The higher the score, the closer the model fragment to the feature description. Therefore, this model fragment would be a better realization of the feature description than any other model fragment with a lower score. Taking into account the scores, the model fragments can be ordered in a ranking where the top positions are occupied by the model fragments with the highest relevance to the feature description. This ranking is the final result of the FLiM-ML approach.

6.4.4 Implementation details

We used the Eclipse Modeling Framework to manipulate the models and CVL to manage the model fragments. For the Machine Learning technique, we selected Rankboost (Freund et al. 2003). RankBoost belongs to the family of Learning to Rank, whose algorithms are specifically designed to perform ranking tasks. Moreover, Rankboost is well known for its efficiency and effectiveness in different domains (Canuto et al. 2013),(Cao et al. 2018). RankBoost was implemented using the RankLib library (Dang 2013) and tuned as in (Kıraç, Aktemur, and Sözer 2018). Specifically, the parameters were tuned with *iteration* = 200 and *metric* equal to ERR10.

6.4.5 Results

In Table 6.1, we outline the mean results for the test cases regarding the configurations (Density-XS, Density-S, Density-M, Density-L, and Density-A) that answer RQ1. Each column shows the Precision, Recall, F-measure, and MCC obtained through each configuration.

In response to RQ1, there are clear differences between the results obtained for the different configurations. In fact, as Table 6.1 shows, Density-XS achieves the best results for all of the performance indicators, providing a

Table 6.1: Mean Values and Standard Deviations for Precision, Recall, F-Measure, and MCC for the test cases regarding the configurations (Density-XS, Density-S, Density-M, Density-L, and Density-A) that answer RQ1

	Precision	Recall	F-measure	MCC
Density-XS	89.74 ± 25.46	82.26 ± 32.11	84.16 ± 29.50	0.81
Density-S	60.73 ± 43.47	58.62 ± 45.21	58.41 ± 44.07	0.47
Density-M	59.17 ± 44.34	57.22 ± 45.32	57.47 ± 44.44	0.48
Density-L	46.74 ± 40.57	41.22 ± 41.35	42.71 ± 40.85	0.28
Density-A	82.06 ± 32.84	74.84 ± 37.50	76.35 ± 35.47	0.72

mean precision value of 89.74%, a recall value of 82.26%, a F-measure value of 84.16%, and a MCC value of 0.81. In contrast, Density-L presents the worst results in all of the indicators, providing a mean precision value of 46.74%, a recall value of 41.22%, a F-measure value of 42.71%, and a MCC value of 0.28. These results indicate that the density values of the model fragments in the knowledge base influence the results. However, the statistical analysis is the final step to claim that the results of the five configurations are significantly different and to assess the magnitude of the improvement from Density-XS.

Therefore, since RQ1 was answered positively, we needed to answer RQ4 and RQ5 for the density property. **In response to RQ4**, Table 6.1 shows that Density-XS achieves the best results for all the performance indicators. Moreover, there is about a 7% of difference between the recall, the precision, and the F-measure obtained for Density-XS and the rest of the configurations. For the MCC, the difference between Density-XS and the rest of the configurations is at least equal to 0.09. These results indicate that the model fragments in the knowledge base should cover the extra-small density values. In other words, the model fragments in the knowledge base should have density values between 0 and 25 in order to obtain the best results.

On the other hand, to answer RQ5, Table 6.2 shows the mean results taking into account the density values of the oracle (approved features) for the test cases. Each column shows the Precision, Recall, F-measure, and MCC obtained through each configuration. Moreover, Figure 6.8 shows the number of approved features in the oracle according to their density values.

In response to RQ5, as Table 6.2 shows, Density-XS achieves the best results for all of the performance indicators regardless of the test cases. In fact, a

Table 6.2: Mean Values and Standard Deviations for Precision, Recall, F-Measure, and MCC for the test cases grouped by the density values of their approved features, regarding the configurations: Density-XS, Density-S, Density-M, Density-L, and Density-A

Density values in test cases	Configuration	Precision	Recall	F-measure	MCC
[0,25)	Density-XS	87.85 ± 27.18	86.83 ± 30.40	86.80 ± 29.08	0.86
	Density-S	59.40 ± 45.88	59.44 ± 47.76	59.11 ± 46.87	0.58
	Density-M	55.76 ± 48.52	56.86 ± 49.00	56.24 ± 48.71	0.55
	Density-L	41.86 ± 44.02	39.75 ± 44.85	40.08 ± 44.55	0.38
	Density-A	85.21 ± 30.43	84.03 ± 32.82	83.87 ± 31.94	0.83
[25,50)	Density-XS	96.30 ± 17.52	90.74 ± 25.16	92.10 ± 21.94	0.92
	Density-S	61.33 ± 39.08	60.62 ± 41.52	59.06 ± 39.57	0.49
	Density-M	45.92 ± 43.43	44.44 ± 44.78	43.99 ± 42.95	0.35
	Density-L	41.96 ± 39.69	36.48 ± 39.83	37.79 ± 39.31	0.28
	Density-A	76.61 ± 40.63	73.33 ± 41.57	73.72 ± 40.59	0.73
[50,75)	Density-XS	75.93 ± 37.71	63.37 ± 35.24	68.16 ± 35.40	0.55
	Density-S	45.85 ± 45.07	44.59 ± 44.44	44.70 ± 44.17	0.20
	Density-M	51.57 ± 43.13	45.06 ± 40.06	47.09 ± 40.33	0.27
	Density-L	42.86 ± 39.96	36.61 ± 35.70	38.64 ± 36.57	0.14
	Density-A	72.84 ± 35.66	59.67 ± 35.88	63.70 ± 34.48	0.49
[75,100]	Density-XS	95.37 ± 11.27	83.49 ± 32.46	85.66 ± 27.67	0.83
	Density-S	72.57 ± 41.26	66.36 ± 45.33	67.36 ± 43.84	0.56
	Density-M	81.44 ± 33.09	79.48 ± 37.48	79.94 ± 35.80	0.70
	Density-L	59.18 ± 36.38	50.84 ± 42.47	53.25 ± 40.57	0.28
	Density-A	91.36 ± 19.08	78.80 ± 35.72	81.15 ± 31.59	0.77

Table 6.3: Mean Values and Standard Deviations for Precision, Recall, F-Measure, and MCC for the test cases for the configurations (Multiplicity=1, Multiplicity>1, and Multiplicity-A) that answer RQ2

	Precision	Recall	F-measure	MCC
Multiplicity=1	91.13 ± 23.44	89.91 ± 26.10	89.86 ± 24.83	0.89
Multiplicity>1	90.26 ± 24.77	87.94 ± 28.26	88.16 ± 26.56	0.88
Multiplicity-A	92.14 ± 21.99	89.97 ± 26.12	90-06 ± 24.25	0.90

knowledge base that is configured using model fragments with extra-small density values (Density-XS) obtained the results for the test cases whose approved features have density values between 0 and 25, providing a mean precision value of 87.85%, a recall value of 86.83%, a F-measure value of 86.80%, and a MCC value of 0.86. The same configuration obtained the best results for the test cases whose approved features have density values between 25 and 50, providing a mean precision value of 96.30%, a recall value of 90.74%, a F-measure value of 92.10%, and a MCC value of 0.92. This configuration also obtained the best results for the test cases whose approved features have density values between 50 and 75, providing a mean precision value of 75.93%, a recall value of 63.37%, a F-measure value of 68.16%, and a MCC value of 0.55. Finally, the same configuration obtained the best results for the test cases whose approved features have density values between 75 and 100, providing a mean precision value of 95.37%, a recall value of 83.49%, a F-measure value of 85.66%, and a MCC value of 0.83. Therefore, although the searched solutions in the test cases have different density values than the model fragments in the knowledge base, the obtained results can be even better than the results obtained using model fragments with similar density values.

In Table 6.3, we outline the mean results for the test cases regarding the configurations (Multiplicity=1, Multiplicity>1, and Multiplicity-A) that answer RQ2. Each column shows the Precision, Recall, F-measure, and MCC obtained through each configuration.

In response to RQ2, the results obtained for all of the configurations are very similar. Therefore, Table 6.3 does not provide enough evidence to say with certainty that multiplicity influences the results. For this reason, RQ2 is answered taking into account the statistical analysis.

In Table 6.4, we outline the mean results for the test cases for the configurations (Dispersion-XS, Dispersion-S, Dispersion-M, Dispersion-L, and Dispersion-A)

Table 6.4: Mean Values and Standard Deviations for Precision, Recall, F-Measure, and MCC for the test cases for the configurations (Dispersion-XS, Dispersion-S, Dispersion-M, Dispersion-L, and Dispersion-A) that answer RQ3

	Precision	Recall	F-measure	MCC
Dispersion-XS	86.84 ± 30.38	86.96 ± 26.29	85.63 ± 28.72	0.85
Dispersion-S	86.79 ± 30.24	87.21 ± 26.56	85.75 ± 28.83	0.85
Dispersion-M	33.61 ± 32.31	34.61 ± 33.88	32.72 ± 32.48	0.28
Dispersion-L	66.58 ± 39.97	67.20 ± 35.98	65.86 ± 37.77	0.64
Dispersion-A	73.73 ± 37.16	78.77 ± 29.27	73.80 ± 33.89	0.73

that answer RQ3. Each column shows the Precision, Recall, F-measure, and MCC obtained through each configuration.

In response to RQ3, there are clear differences between the results obtained for the different configurations. In fact, as Table 6.1 shows, Dispersion-XS and Density-S achieve the best results: Density-XS attains 86.84% precision, 86.96% recall, 85.63% F-measure, and 0.85 MCC; and Dispersion-S attains 86.79% precision, 87.21% recall, 85.75% F-measure, and 0.85 MCC. In contrast, Dispersion-M presents the worst results in all of the indicators, providing a mean precision value of 33.61%, a recall value of 34.61%, a F-measure value of 32.72%, and a MCC value of 0.28. These results indicate that the dispersion of the model fragments in the knowledge base influences the results. However, the statistical analysis is the final step to claim that the results of the five configurations are significantly different and to assess the magnitude of the improvement from Density-XS or Density-S.

Therefore, since RQ3 was answered positively, we needed to answer RQ4 and RQ5 for the dispersion property. **In response to RQ4**, Table 6.4 shows that Dispersion-XS and Dispersion-S achieve the best results for all of the performance indicators. Moreover, the difference between the recall, the precision, and the F-measure obtained for these two configurations and the rest of configurations is greater than 12%. For the MCC, the difference between these two configurations and the rest of the configurations is at least equal to 0.12. To all appearances, these results indicate that the model fragments in the knowledge base should cover at least one of these configurations: extra-small dispersion values or small dispersion values. In other words, the model fragments in the knowledge base should have dispersion values between 0 and 25 or between 25 and 50 in order to obtain the best results. However, a statistical analysis can

Table 6.5: Mean Values and Standard Deviations for Precision, Recall, F-Measure, and MCC for the test cases grouped by the dispersion values of their approved features regarding the configurations: Dispersion-XS, Dispersion-S, Dispersion-M, Dispersion-L, and Dispersion-A

Dispersion values in test cases	Configuration	Precision	Recall	F-measure	MCC
[0,0.25)	Dispersion-XS	100.00 ± 0.00	88.56 ± 7.79	93.76 ± 4.25	0.93
	Dispersion-S	100.00 ± 0.00	89.55 ± 8.12	94.30 ± 4.43	0.94
	Dispersion-M	30.27 ± 12.57	17.41 ± 11.76	21.75 ± 11.37	0.14
	Dispersion-L	100.00 ± 0.00	88.56 ± 7.79	93.76 ± 4.25	0.93
	Dispersion-A	100.00 ± 0.00	89.30 ± 8.05	94.17 ± 4.39	0.94
[0.25,0.50)	Dispersion-XS	95.24 ± 15.82	93.37 ± 20.20	93.68 ± 18.84	0.93
	Dispersion-S	95.14 ± 15.07	92.87 ± 21.60	93.43 ± 19.58	0.92
	Dispersion-M	54.04 ± 44.00	51.14 ± 45.02	51.79 ± 44.75	0.46
	Dispersion-L	77.11 ± 36.41	73.51 ± 40.10	74.69 ± 38.64	0.71
	Dispersion-A	94.89 ± 15.98	92.44 ± 21.65	93.02 ± 19.96	0.92
[0.50,0.75)	Dispersion-XS	73.96 ± 42.87	74.13 ± 42.36	73.82 ± 42.44	0.73
	Dispersion-S	74.10 ± 42.54	74.63 ± 42.49	74.14 ± 42.27	0.73
	Dispersion-M	23.66 ± 37.42	27.36 ± 36.55	24.88 ± 36.71	0.21
	Dispersion-L	58.21 ± 49.50	56.72 ± 49.06	57.16 ± 49.04	0.54
	Dispersion-A	74.23 ± 42.73	75.12 ± 42.82	74.14 ± 42.44	0.72
[0.75,1]	Dispersion-XS	78.17 ± 34.87	91.79 ± 18.78	81.28 ± 30.45	0.82
	Dispersion-S	77.92 ± 35.13	91.79 ± 18.78	81.13 ± 30.63	0.81
	Dispersion-M	26.46 ± 13.39	42.54 ± 21.91	32.47 ± 16.40	0.30
	Dispersion-L	31.00 ± 8.22	50.00 ± 15.19	37.81 ± 9.60	0.37
	Dispersion-A	25.81 ± 9.30	58.21 ± 20.72	33.86 ± 8.85	0.35

determine whether or not there are significant differences between these two configurations (Dispersion-XS and Dispersion-S).

On the other hand, to answer RQ5, Table 6.5 shows the mean results taking into account the dispersion values of the oracle (approved features) for the test cases. Each column shows the Precision, Recall, F-measure, and MCC obtained through each configuration. Moreover, Figure 6.8 shows the number of approved features in the oracle according to their dispersion values.

In response to RQ5, Table 6.5 shows that four configurations (Dispersion-XS, Dispersion-S, Dispersion-L, and Dispersion-A) achieve similar results for the test cases where the searched solutions have dispersion values between 0 and 0.50. Moreover, three configurations (Dispersion-XS, Dispersion-S, and

Dispersion-A) achieve similar results for test cases where the searched solutions have dispersion values between 0.75 and 1 or between 0.50 and 0.75. Finally, for the test cases where the searched solutions have dispersion values between 0.75 and 1, two configurations (Dispersion-XS and Dispersion-S) achieve similar results. Therefore, for the dispersion property, there is more than one configuration that obtain the best results for test cases where the searched solutions have different dispersion values than the model fragments in the knowledge base.

6.4.6 Statistical Analysis

A statistical test must be run to assess whether there is enough empirical evidence to claim that there is a difference between two approaches (e.g., A is better than B). To achieve this, two hypotheses are defined: the null hypothesis H_0, and the alternative hypothesis H_1. The null hypothesis H_0 is typically defined to state that there is no difference between the approaches, whereas the alternative hypothesis H_1 states that the configurations differ. In such a case, a statistical test aims to verify whether the null hypothesis H_0 should be rejected.

Statistical tests provide a probability value, $p-Value$. The $p-Value$ obtains values between 0 and 1. The lower the $p-Value$ of a test, the more likely that the null hypothesis is false. It is accepted by the research community that a $p-Value$ under 0.05 is statistically significant (Arcuri and Briand 2014), and so the hypothesis H_0 can be considered false.

The test carried out depends on the properties of the data. Since our data does not follow a normal distribution in general, our analysis required the use of nonparametric techniques. There are several tests for analyzing this kind of data; however, the Quade test is the most powerful one when working with real data (García et al. 2010). In addition, according to Conover (Conover 1999), the Quade test is the one that has shown the best results for a low number of configurations (no more than 4 or 5 configurations).

Table 6.6 shows the Quade test statistic and $p-Values$ for precision, recall, F-measure, and MCC. For density and dispersion, the $p-Values$ are smaller than 0.05, so we could reject the null hypothesis. In contrast, for multiplicity, the $p-Values$ are not smaller than 0.05, so we could not reject the null hypothesis. Consequently, we can state that there are significant differences among the five configurations for density and among the five configurations for dispersion. However, there are no significant differences among the three

Table 6.6: Quade test statistic and $p - Values$

		Precision	Recall	F-Measure	MCC
Density	p-Value	$< 2.20 \times 10^{-16}$	$< 2.20 \times 10^{-16}$	$< 2.20 \times 10^{-16}$	$< 2.20 \times 10^{-16}$
	Statistic	79.79	63.62	71.41	65.50
Multiplicity	p-Value	0.41	0.15	0.35	0.51
	Statistic	0.87	1.86	1.06	0.67
Dispersion	p-Value	$< 2.20 \times 10^{-16}$	$< 2.20 \times 10^{-16}$	$< 2.20 \times 10^{-16}$	$< 2.20 \times 10^{-16}$
	Statistic	118.7	126.6	111.89	117.41

configurations for multiplicity. Therefore, in response to RQ1, the statistical analysis determines that the density influences the results. In response to RQ2, the statistical analysis determines that the multiplicity does not influence the results. In response to RQ3, the statistical analysis determines that the dispersion influences the results.

Nevertheless, with the Quade test, we cannot answer the following questions: *Which of the configurations regarding density gives the best performance?* and *Which of the configurations regarding dispersion gives the best performance* In this case, the performance of each configuration should be individually compared with all of the other alternatives. In order to do this, we performed an additional post hoc analysis. This kind of analysis performs a pair-wise comparison among the results of each configuration, determining whether statistically significant differences exist among the results of a specific pair of configurations.

Table 6.7 shows the $p - Values$ of Holm's post hoc analysis for each specific pair of configurations according to the RQs. Most of the $p - Values$ shown in this table are smaller than 0.05, so these comparisons have significant differences for all of the performance measurements. However, Table 6.7 shows that there are no significant differences between Density-S and Density-M, between Dispersion-XS and Dispersion-S, and between Dispersion-L and Dispersion-A.

Table 6.7: Holm's Post Hoc $p - Values$

		Precision	Recall	F-Measure	MCC
Density	Density-XS vs Density-S	$< 2.0 \times 10^{-16}$	5.8×10^{-11}	7.6×10^{-14}	1.4×10^{-13}
	Density-XS vs Density-M	$< 2.0 \times 10^{-16}$	1.4×10^{-12}	1.0×10^{-14}	4.9×10^{-15}
	Density-XS vs Density-L	$< 2.0 \times 10^{-16}$	$< 2.0 \times 10^{-16}$	$< 2.0 \times 10^{-16}$	$< 2.0 \times 10^{-16}$
	Density-XS vs Density-A	1.7×10^{-3}	9.7×10^{-3}	7.1×10^{3}	3.9×10^{-3}
	Density-S vs Density-M	0.37	0.5	0.64	1.0
	Density-S vs Density-L	1.4×10^{-5}	4.9×10^{-8}	6.3×10^{-7}	9.8×10^{-7}
	Density-S vs Density-A	4.7×10^{-12}	6.4×10^{-8}	8.1×10^{-10}	3.5×10^{-10}
	Density-M vs Density-L	1.1×10^{-7}	2.2×10^{-9}	9.8×10^{-9}	1.0×10^{-7}
	Density-M vs Density-A	$< 2.0 \times 10^{-16}$	1.6×10^{-11}	2.0×10^{-13}	5.4×10^{-16}
	Density-L vs Density-A	$< 2.0 \times 10^{-16}$	$< 2.0 \times 10^{-16}$	$< 2.0 \times 10^{-16}$	$< 2.0 \times 10^{-16}$
Dispersion	Dispersion-XS vs Dispersion-S	0.99	0.49	0.61	0.6
	Dispersion-XS vs Dispersion-M	$< 2.0 \times 10^{-16}$	$< 2.0 \times 10^{-16}$	$< 2.0 \times 10^{-16}$	$< 2.0 \times 10^{-16}$
	Dispersion-XS vs Dispersion-L	2.4×10^{-10}	1.6×10^{-13}	6.2×10^{-10}	2.8×10^{-10}
	Dispersion-XS vs Dispersion-A	1.0×10^{-11}	1.1×10^{-6}	1.2×10^{-8}	$< 2.7 \times 10^{-9}$
	Dispersion-S vs Dispersion-M	$< 2.0 \times 10^{-16}$	$< 2.0 \times 10^{-16}$	$< 2.0 \times 10^{-16}$	$< 2.0 \times 10^{-16}$
	Dispersion-S vs Dispersion-L	8.7×10^{-11}	3.1×10^{-14}	7.2×10^{-11}	2.1×10^{-11}
	Dispersion-S vs Dispersion-A	2.3×10^{-10}	1.4×10^{-8}	9.7×10^{-10}	4.4×10^{-10}
	Dispersion-M vs Dispersion-L	$< 2.0 \times 10^{-16}$	$< 2.0 \times 10^{-16}$	$< 2.0 \times 10^{-16}$	$< 2.0 \times 10^{-16}$
	Dispersion-M vs Dispersion-A	$< 2.0 \times 10^{-16}$	$< 2.0 \times 10^{-16}$	$< 2.0 \times 10^{-16}$	$< 2.0 \times 10^{-16}$
	Dispersion-L vs Dispersion-A	0.56	2.7×10^{-5}	0.38	0.068

6.4.7 Effect Size

Statistically significant differences can be obtained even if they are so small as to be of no practical value (Arcuri and Briand 2014). It is then important to assess whether a configuration is statistically better than another and to assess the magnitude of the improvement. *Effect size* measures are needed to analyze this.

For a non-parametric effect size measure, we used Vargha and Delaney's $\hat{A}_{12}$ (Vargha and Delaney 2000). $\hat{A}_{12}$ measures the probability that running one configuration yields higher values than running another configuration. If the two configurations are equivalent, then $\hat{A}_{12}$ will be 0.5.

For example, $\hat{A}_{12} = 0.7$ means that we would obtain better results in 70% of the runs with the first of the pair of configurations that have been compared, and $\hat{A}_{12} = 0.3$ means that we would obtain better results in 70% of the runs with the second of the pair of configurations that have been compared. Thus, we have an $\hat{A}_{12}$ value for every pair of configurations.

Table 6.8: $\hat{A}_{12}$ statistic for each pair of configurations

		Precision	Recall	F-Measure	MCC
Density	Density-XS vs Density-S	0.68	0.64	0.66	0.66
	Density-XS vs Density-M	0.69	0.65	0.67	0.67
	Density-XS vs Density-L	0.77	0.76	0.76	0.76
	Density-XS vs Density-A	0.56	0.55	0.55	0.55
	Density-S vs Density-M	0.51	0.52	0.51	0.50
	Density-S vs Density-L	0.58	0.59	0.58	0.59
	Density-S vs Density-A	0.37	0.40	0.39	0.38
	Density-M vs Density-L	0.57	0.57	0.57	0.59
	Density-M vs Density-A	0.36	0.39	0.38	0.37
	Density-L vs Density-A	0.28	0.29	0.29	0.28
Dispersion	Dispersion-XS vs Dispersion-S	0.50	0.50	0.50	0.50
	Dispersion-XS vs Dispersion-M	0.84	0.84	0.84	0.84
	Dispersion-XS vs Dispersion-L	0.63	0.67	0.65	0.65
	Dispersion-XS vs Dispersion-A	0.60	0.59	0.60	0.60
	Dispersion-S vs Dispersion-M	0.84	0.84	0.84	0.84
	Dispersion-S vs Dispersion-L	0.63	0.67	0.65	0.66
	Dispersion-S vs Dispersion-A	0.59	0.60	0.60	0.60
	Dispersion-M vs Dispersion-L	0.28	0.27	0.27	0.26
	Dispersion-M vs Dispersion-A	0.24	0.19	0.22	0.20
	Dispersion-L vs Dispersion-A	0.46	0.41	0.45	0.44

Table 6.8 shows the values of the effect size statistics between every pair of configurations. **In response to RQ4 regarding density**, Density-XS obtains the best results, and Table 6.8 shows the superiority of this configuration in comparison to the others. Specifically, the $\hat{A}_{12}$ measure indicates that Density-XS would obtain better results than Density-S in 68% of the runs for precision, in 64% of the runs for recall, in 66% of the runs for F-measure, and in 66% of the runs for MCC. Density-XS would obtain better results than Density-M in 69% of the runs for precision, in 65% of the runs for recall, in 67% of the runs for F-measure, and in 67% of the runs for MCC. Density-XS would obtain better results than Density-L in 77% of the runs for precision, in 76% of the runs for recall, in 76% of the runs for F-measure, and in 56% of the runs for MCC. Density-XS would obtain better results than Density-S in at least 55% of the runs for precision, in 55% of the runs for recall, in at least 66% of the runs for F-measure, and in at least 55% of the runs for MCC.

Table 6.8 shows the values of the effect size statistics between every pair of configurations. **In response to RQ4 regarding dispersion**, Dispersion-XS and Dispersion-S obtain the best results. Moreover, according to the post hoc analysis, there are no significant differences between these configurations. In fact, Table 6.8 shows that they are equivalent because all of the $\hat{A}_{12}$ values for all of the indicators are equal to 0.5. The table also shows the superiority of these configurations in comparison to the others. Specifically, the $\hat{A}_{12}$ measure indicates that Dispersion-XS and Dispersion-S would obtain better results than Dispersion-M in 84% of the runs for precision, in 84% of the runs for recall, in 84% of the runs for F-measure, and in 84% of the runs for MCC. Dispersion-XS and Dispersion-S would obtain better results than Dispersion-L in 63% of the runs for precision, in 67% of the runs for recall, in 65% of the runs for F-measure, and in at least 65% of the runs for MCC. Dispersion-XS and Dispersion-S would obtain better results than Dispersion-A in at least 59% of the runs for precision, in at least 59% of the runs for recall, in 60% of the runs for F-measure, and in 60% of the runs for MCC.

6.5 Discussion

The results highlight that the Density-XS configuration achieves better results than any other configuration regarding density. There is not a significant difference between the configurations regarding multiplicity. The Dispersion-XS and Dispersion-S configurations achieve better results than the other configurations regarding dispersion and there is not a significant difference between Dispersion-XS and Dispersion-S.

The first aspect of the results that can be discussed is the fact that none of the model fragment properties must be balanced in order to obtain the best feature location results. The density requires extra-small values, the multiplicity does not matter, and the dispersion requires extra-small or small values. This may be surprising since it is conventional wisdom that an imbalanced knowledge base tends to produce worse results. Therefore, we might hope that the configurations of the knowledge base with balanced model fragment properties (Density-A, Multiplicity-A, and Dispersion-A) we produce the best results. In fact, when this research started, three domain experts discussed if whether any of the three model fragment properties should be used to report the knowledge base. None of the domain experts considered to configure the knowledge base with model fragments that only have some specific property values because this means that the knowledge base would be imbalanced for that property. However, taking into account our results, none of the three properties had to be balanced for our case study. Although these are surprising results, they are indeed positive and beneficial for reducing costs.

The greater the number of properties that have to be balanced, the more expensive and complex the generation of the knowledge base will be. For example, given a knowledge base where half of the samples have a score value equal to *0* and half of the samples have a score value equal to *1*, if the knowledge base had to be balanced for both the score values (*0* or *1*) and also the density values (*<=50* or *>50*), the knowledge base would have to have to be the same number of samples for all of the possible combinations between the score values and the density values: samples with a score value equal to *0* and density value *<=50*, samples with a score value equal to *0* and density value *>50*, score value equal to *1* and density value *<=50*, and samples score value equal to *1* and density value *>50*. In this example, there are only four combinations, but the problem appears when the properties have more than two values and there are more than two properties to be balanced. Therefore, since for our case none of the properties has to be balanced, we only have to balance the scores, and therefore the cost is affordable. Nevertheless, if several properties had to be balanced, we would have to think if the improvement of the results compensates for the cost.

The second aspect of the results that can be discussed is the relation among the configurations.

With regard to density, the model fragments with extra-small density values achieve results with up to 43% more precision, 41% more recall, 42% more F-measure, and 0.53 more MCC than the model fragments with other density values. However, the improvement in the results is progressive. The lower the density, the better the results are. Density-L obtains the worst result. Density-M obtains better results than Density-L. Density-S obtains better results than Density-M. Finally, Density-XS obtain better results than Density-S. For Density-A, that configuration has model fragments for all density values, so it obtains better results than Density-L but it obtains worse results than Density-XS. Table 6.1 and Table 6.2 show this progressive improvement in the results.

It can be observed that this progression is very logical. With extra-small model fragments, we can compose a bigger model fragment by joining several extra-small model fragments like pieces of a puzzle. No matter what density value of the model fragment search for is, the machine learning classifier can locate a small, medium, or large model fragment, locating each extra-small model fragment that composes the small, medium, or large model fragment searched.

Therefore, the knowledge base with extra-small model fragments has enough information to locate any model fragment regardless of its density value. Likewise, the small model fragments can be joined to compose medium and large model fragments. In this case, the machine learning classifier may have problems locating extra-small model fragments because it is not possible to divide small model fragments into extra-small model fragments. For this reason, the configuration of the knowledge base with small model fragments (Density-S) does not obtain better results than the configuration of the knowledge base with extra-small density model fragments (Density-XS). The same behaviour is shown for the results of Density-M and Density-L configurations.

With regard to multiplicity, the three different configurations of the knowledge base no significant differences. This may be due to the definition of the property. The multiplicity property relates a model fragment and a model, determining how many times the model fragment appears in the model. In contrast, both the density property and the dispersion property involve not only the model fragment and model, but also the elements of the model fragment. The density property determines the relation between the number of elements in the model fragment and the number of elements in the model. The dispersion determines the number of elements in the model fragment that are connected among them. Therefore, since the multiplicity property does not involve the content (elements) of the model fragment, the multiplicity property may not help to differentiate one model fragment from another one. For this reason, this property may not be significant from the point of view of the knowledge base.

With regard to dispersion, the model fragments with extra-small and small dispersion values achieve results with up to 53% more precision, 52% more recall, 52% more F-measure, and 0.57 more MCC than the model fragments with other dispersion values. Although the relation between the results is not as clear as in the case of the density property, the stronger the connection is between the lements of a model fragment, the easier it is to find that model fragment. In fact, four different configurations (Dispersion-XS, Dispersion-S, Dispersion-L, and Dispersion-A) achieve good results when the model fragments searched for are strongly connected, so the searched model fragments have values between 0 and 0.25. The same behaviour is shown for the rest of the results; the less connected the model fragments are, the better results the lower configurations obtain. Therefore, there may be a relation between the connected elements in a model fragment and the ease with which the model fragment is located. However, future research about dispersion could help to clarify this relation.

6.6 Threats to Validity

In this section, we use the classification of threats to validity of (Wohlin et al. 2012) to acknowledge the limitations of our experiment.

Construct validity: This aspect of validity reflects the extent to which the operational measures that are studied represent what the researchers have in mind. To minimize this risk, our evaluation is performed using four measures: precision, recall, F-measure, and MCC. These measures are widely accepted in the software engineering research community.

Internal Validity: This aspect of validity is of concern when causal relations are examined. There is a risk that the factor being investigated may be affected by other neglected factors. To reduce this threat, the knowledge base of our case study is large enough to generate suitable views for all of the properties: (density, multiplicity, and dispersion), so the views of the knowledge base are not negatively affected due to imbalance. Moreover, all of the views of the knowledge base contain a similar number of samples (about 1600), so that all of the views are compared taking into account the same conditions.

External Validity: This aspect of validity is concerned with to what extent it is possible to generalize the findings, and to what extent the findings are of relevance for other cases. We reduced this threat by using formats that are frequently leveraged to specify all kinds of different software, for example, MOF. Moreover, our experiment does not rely on the specific conditions of our domain feature descriptions and models. Nevertheless, the experiment and its results should be replicated in other domains before assuring their generalization.

Reliability: This aspect is concerned with to what extent the data and the analysis are dependent on the specific researchers. To reduce this threat, most of the inputs were provided by our industrial partner. Moreover, the only input that was not provided by our industrial partner consisted of the configurations. These configurations were specifically defined to answer the research questions independently of the content of the knowledge base. Moreover, to prevent the researchers from influencing the results by looking for a specific outcome, all of the test cases were evaluated for all of the defined configurations; none of the test cases were removed for any reason whatsoever.

6.7 Related Work

Some existing works focus on machine learning for feature location. For instance, Corley et al. (Corley, Damevski, and Kraft 2015) explore the use of deep learning applied to feature location by the usage of document vectors. Tien-Duy et al. (B. Le et al. 2016) focus on learning to rank through feature vectors that are based on likely invariants. The authors in (Ye, Bunescu, and C. Liu 2014) focus on terms that are defined in a vocabulary to build the feature vectors. Marcen et al. (Ana C. Marcén et al. 2017) presented a feature location approach that targets model fragments as the feature realization artifacts using learning to rank. In (Ana C Marcén, Pérez, and Cetina 2017), the authors propose an evolutionary ontological encoding approach to enable machine learning techniques to be used to perform Software Engineering tasks in models. All of these works propose new approaches to address machine learning techniques in feature location, reporting algorithms, and how to tune them. Our work is focused on reporting the importance of the feature properties when we are training the knowledge base classifier.

Recent years have seen an increasing interest in traceability-based machine learning approaches. Binkley et al. (Binkley and Lawrie 2014) further illustrate the benefits of using the learning-to-rank technique in the context of traceability by applying learning-to-rank algorithms to improve several feature models for software maintenance. In (Falessi et al. 2017), the authors used machine learning classifiers to estimate the number of valid links remaining in a set of candidate links that are returned by IR techniques. More recently, Mills et al. (Mills, Escobar-Avila, and Haiduc 2018) propose TRAIL, a technique for automating traceability maintenance by considering TLR as a binary classification problem. They address this problem using machine learning algorithms that are trained on historical traceability information. In contrast to their works, our work has been evaluated in an industrial domain with SPL engineers by suggesting feature properties to be considered during classifier training.

Several research studies apply machine learning approaches to bug location. For instance, Ye et al. (Ye, Bunescu, and C. Liu 2014) propose a learning-to-rank approach for information retrieval (IR) based on bug localization using features extracted from textual bug reports and source code files. In (Shi et al. 2017), eight learning-to-rank techniques in bug localization are compared. The features are selected from previous hybrid bug localization studies, and the feature weight values in learning-to-rank techniques are learned from historical bug data and source code information. Zhao et al. (Zhao et al. 2015) defined

Table 6.9: Overview of the related work regarding the industrial evaluation and the properties of the knowledge base: size, class distribution (CD), density (DE), multiplicity (MU), and dispersion (DI)

	Related Works	Properties					Industrial
		Size	CD	DE	MU	DI	Evaluation
Feature Location	(Corley, Damevski, and Kraft 2015)	√	X	X	X	X	X
	(B. Le et al. 2016)	√	X	X	X	X	X
	(Ye, Bunescu, and C. Liu 2014)	X	X	X	X	X	X
	(Ana C. Marcén et al. 2017)	√	√	X	X	X	√
	(Ana C Marcén, Pérez, and Cetina 2017)	√	√	X	X	X	√
	(Vale and Santana de Almeida 2018)	X	X	X	X	X	X
Bug Location	(Binkley and Lawrie 2014)	√	X	X	X	X	X
	(Falessi et al. 2017)	√	√	X	X	X	X
	(Mills, Escobar-Avila, and Haiduc 2018)	√	X	X	X	X	X
Traceability Links Recovery	(Ye, Bunescu, and C. Liu 2014)	X	X	X	X	X	X
	(Shi et al. 2017)	√	X	X	X	X	X
	(Zhao et al. 2015)	√	X	X	X	X	X
Our work		√	√	√	√	√	√

three effort-aware metrics, which are all based on lines of code, to examine the actual performance of learning-to-rank bug localization. The authors claimed that the learning-to-rank method is similar or even worse than the standard vector support machine (VSM). All of these works focus on locating bugs by machine learning at the code level; in contrast, our work focuses on the model level.

In summary, our work differs from the previous ones in four aspects: 1) we do not propose a new approach, but rather we focus on assessing the impact of feature properties in the training of the classifier; 2) we evaluate our work in an industrial domain, taking advantage on domain knowledge; 3) we focus our efforts in a model-based approach; and 4) when training classifiers, we are the only ones who have carefully evaluated the impact of feature properties on the quality of the results.

6.8 Conclusions

Feature location (FL) is the task of finding the features that implement a specific, user-observable functionality in a software system. It plays a key role in Software Product Lines initiation when products already exist. The emerging interest of leveraging machine learning to address the challenge of feature location is rapidly growing (Ana C Marcén, Pérez, and Cetina 2017),(Ana C. Marcén et al. 2017),(Corley, Damevski, and Kraft 2015),(Binkley and Lawrie 2014),(Mills, Escobar-Avila, and Haiduc 2018),(B. Le et al. 2016). Significantly, machine learning techniques are presented as an interesting perspective for automatically locating features. To do this, the knowledge and the experience of the companies are collected in knowledge bases so that the machine learning techniques can benefit from this information for feature location.

Nevertheless, feature location on models involves specific properties, which may not be relevant in other domains. Although most of the works report the machine learning techniques, the tuning parameters, and the size and class distribution of the knowledge base, they do not discuss the model fragment properties. In models, the knowledge base contains model fragments, whose properties (density, multiplicity, and dispersion) may or may not influence the feature location results. Therefore, since model fragments make up part of the knowledge base, they should be properly reported in case they have an impact on the feature location results.

In this paper, we have analyzed the influence of three model fragment properties: density, multiplicity, and dispersion. After the evaluation in the industrial domain of train control firmware, our results show that the density and dispersion properties significantly influenced the results in our case study. In contrast, the multiplicity property did not influence the results, no matter which multiplicity values of the model fragments were used. Based on the results of our case study, for future reports, the model fragments in the knowledge base regarding the density and dispersion properties should be described. Likewise, works on machine learning-based feature location on models should also analyze the influence of model fragment properties on their case studies not only to properly report but also to compare the approaches fairly, thereby improving the feature location results of their case studies.

Part III

Discussion

Discussion

Through the pages of this chapter, we discuss the rationale for the development of the thesis. We also discuss the results of our work by connecting the research questions posed at the start of the thesis with the research articles that conform the compendium included in chapters 1 to 6. In addition, we describe our ongoing research and ideas for future works.

Thesis Development Rationale

Not so very long ago, the Software Engineering Community addressed activities such as Feature Location or Traceability Links Recovery solely on code. We could identify a shortcoming with regards to the available research on the topic within artifacts other than code. In particular, we identified a lack of research with regards to model-based software artifacts in general, and more specifically, within the context of retrieving model-based artifacts.

As mentioned above, search problems in models are relatively new when compared to search problems in other kinds of documents such as the web or source code. This has led the Model-Based Software Engineering Community to invest efforts and resources in the growth of this area. In line with this situation, we invested our efforts and initiatives with the goal of trying to make our contribution there.

Therefore, our first efforts were aimed at transforming the state-of-the-art approaches in code-based software artifacts, so that they could be transported to model-based software artifacts. We focused our efforts on automating the variability formalization of a given family of products into a Model-based Software Product Line (SPL). We published this work in the 3rd International Workshop on REverse Variability Engineering (REVE SPLC '15), chapter 1. After that, we were able to rank the relevancy of legacy products for a new development at the requirements level, and to locate their most significant methods for each of the new product requirements. Focusing our efforts on a new artifact level, the requirements level, we published a paper for the 20th International Systems and Software Product Line Conference (SPLC '16), chapter 2. As a result of this work, we suggested to software engineers the best way of locating the most relevant methods to each requirement.

Afterwards, we focused on abstracting the clone-and-own relationships of different families of Model-based software products. We successfully leveraged feature location at the model level and code comparisons in order to identify the clone-and-own relationships between the same features in different model-based product variants. This work was published in the 15th International Conference on Software Reuse (ICSR '16), chapter 3 and later improved, extended, and published in IEEE Acess '18, chapter 4.

While our initial works were substantially successful, we identified a series of problems related to how the location problems were reporting their results. In particular, we realized that our initial works were not reporting the case studies experiments deeply, since we were considering the model size as the only measurement for reporting the results of our experiments. This problem extended to most works by the research community.

Going on, we identified a series of facts that served as a starting point for discussing why more measurements should be considered for reporting location problems. There is an inability of the reported measurements to accurately represent the inherent challenge posed by the model level. We were able to identify a set of five measurements which should be considered during the report of location problems. We propose the usage of the model size and the usage of the model volume as measurements to report the search space (models), and the usage of density, multiplicity, and dispersion as measurements to report the solution space (model fragments). Size and volume are descriptive measurements. Density, multiplicity, and dispersion are diagnostic measurements. Both descriptive measurements and diagnostic measurements look to the past to explain what happened and why it happened. We evaluated our proposal pushing it to industrial settings, using an Information retrieval real-world approach based on feature location. This work was published in the 21st International Conference on Model Driven Engineering Languages and Systems (MODELS '18), chapter 5.

At this point, we recapped our work, integrating our measurements on a Machine Learning approach for feature location. We were able to demonstrate on an industrial environment that the model size measurement (widespread measurement so far) is not the only one that should be considered in order to report location problems, being other measurements such as density and dispersion needed to deeply report the location problems during MFL. This work was published in a paper for the Information and Software Technology journal (IST '20), chapter 6.

Finally, I built this book, and commenced working towards the rest of the ideas and issues exposed by my previous research. The results of this discussion highlight that ours is a promising work that opens the door for the study of the particularities of reporting the location problems of Model Fragment Location.

Research Questions

The two research questions posed at the introduction of this thesis have been tackled throughout the research works presented in chapters 1 to 6. In the following paragraphs, we present how each of the research articles has contributed to the study of the research questions:

RQ1 Determine whether, and to what extent, our proposed measurements influence Information Retrieval (IR) Model Fragment Location approaches.

Response to RQ1 : All of the previous works use at least one Model Fragment Location technique, but only the last one discusses the need of deeply improving the task of reporting location problems. The results show that all of the proposed measurements (size, volume, density, multiplicity, and dispersion) have a direct impact on the results of the MFL approaches. All of this points towards the fact that the proposed measurements and the provided values are significant for the MFL research community.

RQ2 Determine whether, and to what extent, our proposed measurements influence Machine Learning-based (ML) Model Fragment Location approaches.

Response to RQ2 : Evaluating the proposed measurements on a Machine Learning-based approach for feature location we have realised that not all of the measurements have a direct impact on the results. Our results show that density and dispersion significantly influence feature location results. By contrast, our results show that the multiplicity does not have an impact on the feature location results. Those Machine Learning-based approaches which use training datasets should describe the model fragments regarding the density and dispersion measurements.

As a result of the experiments we carried out, we were able to discuss why the proposed measurements and the provided values are significant for the research community. Regarding the results obtained by applying our measurements to an Information Retrieval-based approach for feature location, it is possible to conclude that size reports do not provide enough information. This is due to the inability of the size measurement to accurately represent the inherent challenge posed by the model level. Also, due to the absence of real work datasets, synthetic datasets are very common and popular in the research community. Since our findings show that the proposed measurements are significant and impactful with regard to the performance of search problems, we provide their real-world values so that designers can carry out a real-world search-problem profiling process when designing synthetic test cases and extreme search-problem scenarios.

On the other side, regarding the results obtained by applying our measurements to a Machine Learning-based approach for feature location, it is possible to conclude that none of the measurements has to be balanced in order to obtain the best feature location results. In addition, another aspect that can be discussed is the relation among the training datasets. With regard to density, the lower the density, the better the results are, being the model fragments with extra-small density values those that achieve best results. With regard to multiplicity, since this measurement does not involve the content (elements) of the model fragment, the multiplicity may not help to differentiate a model fragment from another one, and consequently, this measurement does not influence the feature location results. Finally, with regard to dispersion, there is a relation between the connected elements in a model fragment and the complexity to locate the model fragment. In other words, the stronger the connection between the elements is, the easier it is to find the model fragment.

By acknowledging and incorporating the proposed measurements in this thesis, the approaches are better equipped to deal with the challenge of Model Fragment Location, and thus obtain deeper results, than those baseline approaches that do not take in account this novel knowledge. Hence, our research proves that the reporting activity of these approaches and techniques can be successfully refined to improve the task under study.

Future Work

This section presents some of the ideas and opportunities for future work that arose from our latest research:

1. The results show that our proposed measurements properly report the location problem. We focused on both descriptive and diagnostic measurements to explain what happened (search space) and why it happened (solution space). However, it is possible to find other types of measurements, such as predictive measurements, which use historical data to forecast what will happen in the future and what actions can be taken to affect those outcomes. Thus, a novel research question arises: *Are there predictive measurements that can exert a positive or a negative influence on feature location?*

   To answer this question, it would be necessary to analyze those measurements which have been accepted by the community in similar fields and then conduct an evaluation to check whether acknowledging these measurements leads MFL approaches to significantly enhanced results.

2. The analysis of the results shows that both the density and the dispersion measurements significantly influence the results on a machine learning-based approach. Thus, a novel research question arises: *Can the same results be obtained through another machine learning-based approach?*

   To answer this question, it would be necessary to devise mechanisms for incorporating those measurements into the new approaches, and to analyze the impact of the proposed measurements on this new developments.

3. In industrial scenarios, the usage of several Architectural Description Languages to specify different software systems is common. Thus, related to works which apply MFL techniques to Architecture Models, the following question arises: *Do the proposed measurements influence the results of Traceability Links Recovery among requirements and architecture models?*

   To answer this question, it would be necessary to take advantage of all of the proposed measurements and use architecture models, which have not taken into consideration so far during Traceability Links Recovery processes, as a valid artifact.

These ideas constitute seeds for a yet uncharted but clear path towards future improvements, and allow us to keep working in a topic that is still an open, interesting, and relevant issue for the Software Engineering community.

Part IV

Conclusions

Conclusions

This chapter presents the final concluding remarks of the thesis, summarizing the challenge, scope, and outcomes of our work.

Nowadays, software exists in almost everything. Companies often develop and maintain a collection of custom-tailored software systems that share some common features but also support customer-specific ones. As the number of features and the number of product variants grows, software maintenance is becoming more and more complex. To keep pace with this situation, the Software Engineering Community is addressing a key-activity: Model Fragment Location (MFL). Model Fragment Location (MFL) aims at identifying model elements that are relevant to a requirement, feature, or bug. Many MFL approaches have been introduced in the last few years to address the identification of the model elements that correspond to a specific functionality. The emerging interest in leveraging machine learning to address the challenge of MFL has been growing rapidly (Corley, Damevski, and Kraft 2015), (B. Le et al. 2016), (Ye, Bunescu, and Liu 2014), (Marcén et al. 2017), (Binkley and Lawrie 2014). Although most of the works report the machine learning techniques, the tuning parameters, and the size and score distribution of the training datasets, they do not discuss the model fragment measurements, with model size being the only reported measure. Since different models are measured in different ways, model size values are not a valid comparison. Nevertheless, despite the popularity of different MFL approaches, the question of how the usage of measurements influences MFL results has not yet received much attention.

In the last decade, different types of measurements have been discussed in order to analyze and report information (descriptive, diagnostic, predictive, and prescriptive measurements, among others) (Delen and Ram 2018). Both descriptive measurements and diagnostic measurements look to the past to explain what happened and why it happened. Predictive measurements and prescriptive measurements use historical data to forecast what will happen in the future and what actions can be taken to affect those outcomes. Through this thesis, we have proposed using five measurements to report the location problems during MFL. Size and volume are descriptive measurements, while density, multiplicity, and dispersion are diagnostic measurements. We have evaluated the influence of these measurements on two different case studies, the first one based on an Information Retrieval-based (IR) approach for feature location, and the second one based on a Machine Learning-based (ML) approach for feature location. To that extent:

1. In the first stages of the thesis, our first efforts were aimed at transforming the state-of-the-art approaches in code-based software artifacts, so that they could be transported to model-based software artifacts. In particular, we worked towards helping software engineers by automating the variability formalization of their software products into a Model-based Software Product Line. In addition, we focused on ranking the relevancy

of legacy products for a new development at the requirements level and to locate their most significant methods for each of the new product requirements. Also, through other works, we identified the clone-and-own relationships that are inherently present across a model-based family of software products.

2. As a result of our initial work, we identified a series of problems related to how the results of the experiments are reported. In particular, our first work was evaluated only taking into account the size of the models. Size was already studied in other research works, in fact, this measurement is frequently used in most MFL works. We noticed that size should not be the only reported measure.

3. Afterwards, we proposed using five measurements for reporting the location problem of MFL. Some of these measurements focus on measuring the search space, and the rest of them focus on measuring the solution to be searched for. Our goal is to determine the relevance of the proposed measurements on the results provided by different Information Retrieval-based (IR) MFL approaches.

4. In the final stages of the thesis, we worked on evaluating the proposed measurements by applying them to a Machine Learning-based (ML) approach for feature location. Some of the measurements significantly influence the results.

Our research has been validated through two real-world industrial case studies. In addition, our research has contributed to several national and international projects. Partial results of our research have been published in several scientific articles in workshops and conferences, relevant in the field of our studies. Overall, the results of our research have concluded that: (1) properly reporting the location problem is important because otherwise it is not possible to compare the performance of different approaches against each other, and (2) taking into account the proposed measurements may be relevant to enable the replication of research works that locate features by applying a machine learning technique. This uncharted territory leaves room for future work in such an interesting and relevant topic.

To close this book, we reflect that, while there is still much work to do in the field, we have conclusively developed a PhD thesis that successfully contributes to advancing the available research knowledge in the Definition of Descriptive and Diagnostic Measurements for Model Fragment Retrieval.