Feature and Bug Localization on Model-based Systems at Design time and Runtime

Lorena Arcega

May 20, 2019

Thesis submitted for the degree of Philosophiæ Doctor

ABSTRACT

FOR more than two decades, the modeling community has made great efforts to address key issues such as a Unified Modeling Language (UML), a formal language to describe expressions on UML (OCL), or model transformation languages. In addition, all software systems evolve over time. In fact, some works point that up to 80% of the lifetime of a system is spent on maintenance and evolution activities. However, the maintenance of software systems built by Model Driven Development (MDD) has not received as much attention as we think it should. Recent surveys of feature and bug localization do not identify a single approach that addresses the models of MDD. Both Feature Localization and Bug Localization are essential tasks in maintenance:

- Feature localization is important because: before modifying the system, it is necessary to locate what is going to be modified, and
- bug localization aims to identify the location in the system that is pertinent to a software fault. Bug localization is one of the most important, expensive, tedious and time-consuming activities in program maintenance.

In this thesis we propose feature localization and bug localization approaches for MDD. In MDD, the main artifact to develop the software is the model and, therefore, it should not be ignored when locating features and bugs. In the case of feature localization, we extend our previous works that had already covered the models at design time, to address the models at runtime. In the case of bug localization, this thesis addresses both design time and runtime.

Our feature localization approach combines architecture models at runtime and Information Retrieval (IR) for feature location. The execution information is collected in the model at runtime. Then, the approach applies an information retrieval technique at model level. The result is a ranked list of model elements that are related to the desired feature based on the similarity to the feature description.

Our bug localization approach covers design time and runtime. Our design time approach is a Multi-Objective Evolutionary Algorithm. This approach searches over model fragments and is guided by the textual similarity to the bug report and by the Defect Localization Principle. Our runtime approach is focused on locating bugs that appear as the result of dynamic reconfigurations of the system due to context changes. Our approach for bug localization in reconfigurations applies an

evolutionary algorithm guided by a fitness function that considers the similarity to the text description of the bug report. The solutions provided by our approach are sequences of reconfigurations that, when followed, might lead to the model at runtime which contains the located bug.

We evaluated our approaches in two real-world industrial case studies: BSH, the leading manufacturer of home appliances in Europe, and CAF, a worldwide leading company that manufactures rolling stock, and in an artificial case study, the Smart Hotel. We measure the results in terms of recall, precision, F-measure and Matthews Correlation Coefficient (MCC). The results of our approaches outperform the baselines that are the approaches used by our industrial partners for feature and bug localization. We also performed statistical analyses to provide evidence of the significance of the results.

This work suggests that the approaches presented in this thesis outperform the current approaches used by the industry that were studied. Specifically, we realize that the models at runtime traces provides useful information that influences the results. In addition, the combination of information retrieval and the Defect Localization Principle leads to a significant improvement when it is applied for bug localization in models. Hence, the study of the reconfiguration of models at runtime helps to locate bugs. Finally, our results show that our approaches can be applied in real world environments, such as BSH and CAF.

Acknowledgements

The work presented in this thesis were carried out at the Department of Informatics, University of Oslo, Norway, and the School of Architecture and Technology, Universidad San Jorge, Spain, during the period 2014-2019. Universidad San Jorge provided me with a research fellowship, for which I am deeply grateful, thank you Pedro Larraz and Luis Correas.

In the first place, I want to thank my two supervisors Øystein Haugen and Carlos Cetina. Thank you for your advice, for sharing interesting conversations and knowledge. Thanks also for encouragement and support all this time.

Thanks to my parents, Félix and Lourdes, for having encouraged, understood and supported me throughout my life. Without your help, I would never have gotten here. Thanks to my grandparents, uncles and cousins because in this life I do not want to spend a whole day without you.

Thanks to my friends for always being there. Special thanks to Ana, Tabuen, Elena, Irene, Lety, Lupe, Noelia, Pilar, Pilola, and Sara. I think that together we form a gear that works quite well. Thanks also to those new friends who are beginning to occupy an important place in my life.

Thanks to the SVIT Research Group. I hope we continue enjoying these 'call for…' in 'Puerto Plaza' or wherever. Thanks to all colleges of the School of Architecture and Technology, each day I learn something new from you. Africa, you have always been a very important support, but much more in this last stage, thank you.

Special thanks to Jaime Font. We have shared this journey together. We have had some very good times and some not so good ones and that has made us a great team. Thanks for the laughter and for everything you teach me. We are no longer co-workers, we are friends.

Thanks to the people from BSH and CAF for sharing their knowledge with us. These allow us to improve our research, test our approaches, and learn how real things work in the industry.

Thanks to the people I met in the conferences for the interesting talks. Thanks to the reviewers for their interesting feedback and suggestions. Finally, special thanks to the members of the adjudication committee.

Lorena Arcega
May 2019

CONTENTS

Abstract iii
Acknowledgements v
Contents xi
List of Figures xiv

Part I Introduction 1

1 Introduction 3
1.1 Motivation 4
1.2 Research Questions 5
1.3 Contribution 6
1.4 Research Methodology 7
1.5 Overview of the Work 8
1.6 Structure of the dissertation 10

2 Background 13
2.1 Overview 14
2.2 Model Driven Engineering 14
2.2.1 Domain Specific Languages 15
2.3 Models at Runtime 16
2.3.1 Autonomic computing and reconfigurations 17
2.4 Information Retrieval 18

3 State of the Art 21
3.1 Overview 22
3.2 Feature Localization at Runtime 23
3.2.1 Motivation of our dynamic feature localization on model approaches 25
3.3 Bug Localization at Design time 25
3.3.1 Motivation of our bug localization on design time model approach 28
3.4 Bug Localization at Runtime 29
3.4.1 Motivation of our bug localization on models at runtime approach 31

Part II Feature and Bug Localization in Models 33

4 Overview of the approaches 35
4.1 Overview . . . 36
4.2 Outline for this dissertation . . . 36
4.3 Running example . . . 38
4.4 Feature and bug examples . . . 40
4.4.1 Feature example . . . 40
4.4.2 Bug example . . . 41
4.4.3 Bug in reconfigurations example . . . 41

5 Dynamic Feature Localization in Models 43
5.1 Overview . . . 44
5.2 Approach . . . 44
5.3 Input . . . 45
5.3.1 Model traces . . . 45
5.4 Population . . . 46
5.5 Search Strategy . . . 46
5.6 Assessment: Information Retrieval . . . 47
5.6.1 Information Retrieval at model trace level . . . 48
5.6.2 Information Retrieval at model level . . . 49
5.7 Output . . . 51

6 Bug Localization in Models with an Evolutionary Algorithm 53
6.1 Overview . . . 54
6.2 Approach . . . 54
6.3 Input . . . 55
6.4 Population . . . 55
6.5 Search Strategy . . . 56
6.5.1 BLiMEA selection . . . 56
6.5.2 BLiMEA crossover . . . 57
6.5.3 BLiMEA mutation . . . 57
6.6 Assessment . . . 59
6.6.1 Model fragment similarity to the bug description . . . 59
6.6.2 The most recent model modification . . . 60
6.7 Output . . . 61

7 Evolutionary Algorithm for Bug Localization in the Reconfigurations of Models at Runtime **63**
7.1 Overview 64
7.2 Approach 64
7.3 Input 65
7.4 Population 65
7.5 Search Strategy 66
7.5.1 EBRo selection 66
7.5.2 EBRo crossover 66
7.5.3 EBRo mutation 68
7.6 Assessment 68
7.7 Output 70

Part III Case Studies and Evaluation **71**

8 Case Studies **73**
8.1 Overview 74
8.2 Induction Hob Domain 74
8.3 Train Control and Management Domain 74
8.4 Smart Hotel Domain 75

9 Evaluations **77**
9.1 Overview 78
9.2 Oracle 78
9.3 Comparison and Measure 79
9.4 Measurements and Statistical Analysis 80
9.4.1 Measurements derived from the comparison 80
9.4.2 Statistical analysis of the measurement results 81
9.5 Results 83
9.5.1 DFL Evaluation 83
9.5.2 BLiMEA Evaluation 85
9.5.3 EBRo Evaluation 86

Part IV Discussion and Conclusion 89

10 Discussion **91**
10.1 Overview 92
10.2 Issues detected in our approaches 94
10.2.1 Vocabulary mismatch 94
10.2.2 Implicit knowledge 94
10.2.3 Invalid reconfiguration sequences 95
10.2.4 Poor use of models 95
10.2.5 Bugs not related to timespan modifications 96
10.2.6 Poor model traces 96
10.3 Findings detected in our approaches 97
10.3.1 Reduction of search space 97
10.3.2 High level of abstraction 97
10.3.3 Good performance between information retrieval and the defect localization principle 98
10.3.4 Non-reliance on the domain 98
10.4 Comparison to other works 98

11 Conclusion **101**
11.1 Overview 102
11.2 Research Questions 102
11.3 Ongoing Research 103
11.4 Related publications 105

Bibliography **106**

Part V Publications 119

12 Dynamic Feature Localization in Models 121
12.1 MRT'15 Paper 123
12.2 MRT'16 Paper 135
12.3 SAM'16 Paper 145
12.4 ECMFA'17 Paper 163
12.5 SANER'16 Paper 181

13 Bug Localization in Models 195
13.1 ISD'17 Paper 197
13.2 MODELS'18 Paper 215
13.3 SOSYM'19 Paper 227

LIST OF FIGURES

1 Introduction **3**
1.1 Our feature and bug localization challenges 6
1.2 Overview of our contributions 9

3 State of the Art **21**
3.1 Overview of the works related to this dissertation 22

4 Overview of the approaches **35**
4.1 Overview of each of the approaches 36
4.2 IHDSL metamodel, syntax, product model, and model fragment realization 39
4.3 Example of a feature 40
4.4 Example of a bug 41
4.5 Example of a context change, a bugged reconfiguration, and the reconfiguration rules performed 42

5 Dynamic Feature Localization in Models **43**
5.1 Overview of DFL 44
5.2 Overview of the Dynamic Feature Location Approach 44
5.3 Different Model Traces following the different Criterion 46
5.4 Information Retrieval via Latent Semantic Indexing (LSI) 50

6 Bug Localization in Models with an Evolutionary Algorithm **53**
6.1 Overview of BLiMEA 54
6.2 The Bug Location Approach in Models: BLiMEA 55
6.3 Crossover and mutation operations of BLiMEA 58
6.4 Timespan of the modifications of the model elements of a fragment 61

7 Evolutionary Algorithm for Bug Localization in the Reconfigurations of Models at Runtime **63**
7.1 Overview of EBRO 64
7.2 Input and output of our bug localization in reconfigurations of models at runtime 65
7.3 Representation of an individual 66
7.4 Crossover and mutation operators applied to reconfigurations . . 67
7.5 Terms extraction from a reconfiguration sequence 69

8 Case Studies **73**
8.1 Smart Hotel Model Reconfigurations 76

9 Evaluations **77**
9.1 Evaluation process for each of the approaches 78

10 Discussion **91**
10.1 Systems in which we have applied our approaches 92
10.2 Application flow for a model-based reconfigurable system . . . 93

Part I

INTRODUCTION

1

INTRODUCTION

Contents

1.1 Motivation **4**
1.2 Research Questions **5**
1.3 Contribution **6**
1.4 Research Methodology **7**
1.5 Overview of the Work **8**
1.6 Structure of the dissertation **10**

1.1 Motivation

Nowadays, software exists in almost everything. This trend has been accompanied by a high increase in the scale and the complexity of software. Model-Driven Engineering (MDE) is being applied in an ever-increasing manner to cope with the complexity of software systems by raising the level of abstraction [1].

Nevertheless, similar to any kind of software, MDE systems need to be maintained and evolved over time. Lehman et al. [2] 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. Feature and bug localizations are two of the most important activities performed by developers during software maintenance and evolution.

To establish what we refer to when we talk about feature and bug localization, we consider the following definitions:

Feature. A feature represents a functionality of a system that is defined by requirements and is accessible to developers and users.

Bug. A bug refers to an error or fault in any system that produces unexpected results or causes a system to behave unexpectedly.

Feature or bug localization. Feature localization techniques aim at locating software artifacts that implement a specific program functionality, i. e., a feature. Similarly to feature localization, bug localization is focused on identifying the locations of software faults, i. e., bugs.

Models. In the model paradigm, models can play several roles in software development: diagrams for analysis, can be reverse-engineered from source code, or can be used for code generation. In addition, models at runtime are defined as causally connected self-representations of the associated system that emphasize the structure, behavior, or goals of the system from a problem space perspective. In this work, we focus on models that can be used for code generation (hereinafter, models at design time) and on models at runtime.

Software maintenance and evolution involves adding new features to programs, improving existing functionalities, and removing bugs. At the end of the process, the software engineer gets a piece of software that is in charge of some

functionality in the system using any of the approaches. However, if we want to perform bug localization, we must take into account different particularities than if we want to perform feature localization.

When we work with models (at design time or runtime), there are no approaches for performing basic maintenance tasks such as feature or bug localization. Most research on feature or bug localization considers code to be the software artifact that realizes the feature or contains the bug, neglecting other software artifacts such as models.

The companies that have adopted the MDE paradigm have the advantage of working at a high level of abstraction. Compared to working at the code level, models provide the advantage of abstracting the implementation details. Nevertheless, source-code-based systems have a large number of approaches to perform maintenance activities, while in the case of model-based systems, recent surveys in feature and bug localization have not identified any. The main goal of this thesis is to contribute to changing this.

Thus, we have identified two main challenges (see Figure 1.2): (1) Dynamic Feature Localization in Models; and (2) Bug Localization in Models. To deal with the first challenge, we present an approach that leverages the use of model at runtime traces to perform dynamic feature location. To deal with the second challenge, we present two approaches that use evolutionary algorithms to perform bug localization in design time and in runtime models.

1.2 Research Questions

Recent surveys [3, 4, 5] reveal that none of the feature or bug localization approaches take into account models for locating features or as the source of the bugs. In the model paradigm, models can play several roles in software development: diagrams for analysis, can be reverse-engineered from source code, or can be used for code generation. In this work, we focus on models for code generation. When models are used for code generation, locating features or addressing bugs at the model level must not be neglected.

Therefore, specific feature and bug localization approaches for model-based systems are necessary. Moreover, since there is a lot of work done for code-based systems, we can study how to apply those techniques to feature and bug localization approaches for model-based systems. In addition, taking into account the issues that our industrial partners have when locating features or bugs, we

have realized that both the design time models and the runtime models must be considered.

In this dissertation, we move in this direction by addressing three research questions related to these challenges:

Research Question 1: How does the use of runtime traces gives valuable additional information on only static program/models in Feature Location?

Research Question 2: To what extent are the techniques for bug localization that are used in program code applicable to software based on models as well?

Research Question 3: How can we best locate bugs in model-based systems that are subject to dynamic reconfigurations?

1.3 Contribution

Figure 1.1 shows a summary of what we have done in this thesis. Our starting point is FLiMEA [6].

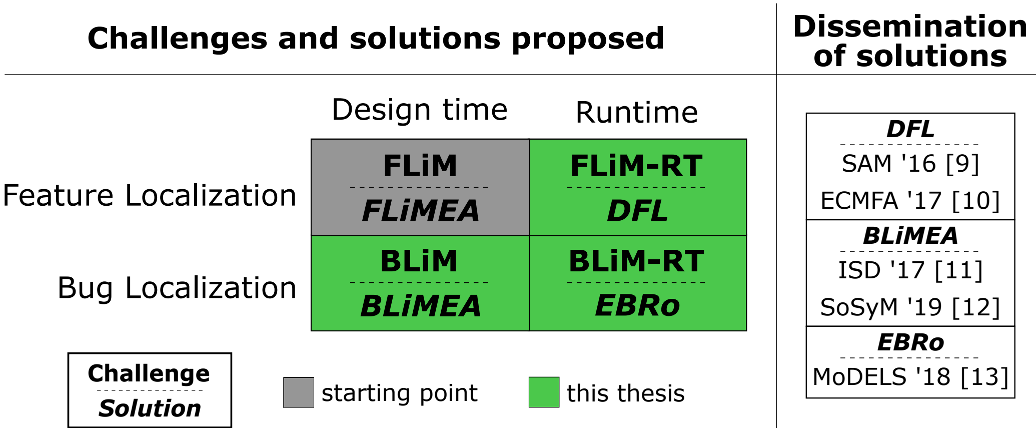


Figure 1.1: Our feature and bug localization challenges

To address Research Question 1, the FLiM-RT challenge, we present Dynamic Feature Localization [7, 8, 9, 10]: an approach for feature localization in models at runtime. We use two different techniques to perform dynamic feature localization: Dynamic Analysis and Information Retrieval. In addition, we study how the information extracted from runtime model traces can be used for feature localization.

To address Research Question 2, the BLiM challenge, we present BLiMEA [11, 12]: an approach for bug localization in models. The approach is a Multi-Objective Evolutionary Algorithm that uses information retrieval and the defect localization principle as fitness functions. First, we evaluate four applications of the defect localization principle. Second, we take advantage of the domain information from the model and the metamodel.

To address Research Question 3, the BLiM-RT challenge, we present EBRo [13]: an approach for bug localization in models at runtime. Specifically, we focus on bugs that appear as the result of dynamic reconfigurations of the system due to context changes. The approach for bug localization in reconfigurations is an evolutionary algorithm. We guide the evolutionary algorithm with a fitness function that measures the similarity to the description of the bug report.

We have also evaluated the presented contributions with our industrial partners, applying them to industrial product models and using the domain knowledge from their domain experts. The contributions have been developed within national and international research projects aligned with the research performed in this dissertation. The contributions have been shared with the research community in the form of conference and peer-reviewed journal publications. Finally, we have identified different challenges that remain unaddressed in this dissertation and that constitute our ongoing research.

1.4 Research Methodology

For the development of this thesis work, we have followed the methodological structure for the development of doctoral theses proposed by Wieringa in [14]. It is a cyclical and iterative methodology and consists of the following phases:

Research problem analysis. In this phase, the main topics to be investigated are searched and the research questions are formulated.

Research design and inference design. In this phase, each one of the empirical studies has been developed in order to answer the research questions formulated.

Validation of research and inference design. We have validated our studies with the datasets from our industrial partners. In addition, these studies have been validated with the publication of the results.

Research execution. Studies in feature and bug localization have been carried out, applying several techniques at both design time and runtime. The result is seven research papers.

Data analysis. Empirical studies have shown results that have allowed us to obtain answers to our research questions. In addition, they indicate the direction to follow in the software maintenance tasks studied at the model level.

The cyclic process described above has been developed iteratively based on the initial empirical study. However, the knowledge produced (the results obtained) has led the research to take into account new perspectives.

Following the cycle defined by Wieringa in [14], the first cycle began with the awareness of the problem in an industrial environment where the software is developed under the MDD paradigm. Initially, the problem to be solved was identified and the study was designed to obtain answers to the associated research questions. The results obtained in this cycle triggered new problems for investigation. These new problems have been designated as starting points for further research works.

In this thesis, we tried to solve the problem of feature and bug localization in model-based systems. The companies that develop systems based on models have the advantages of working at a high level of abstraction compared to working at the code level. However, the disadvantage is that there are no approaches to perform maintenance activities such as feature or bug localization. Recent surveys in feature and bug localization [4, 5] do not identify any. The main objective of this thesis is to contribute in developing solutions to perform feature and bug localization. Finally, the artifacts produced in this thesis are three approaches for feature and bug localization based on models that takes into account the techniques used in program code based approaches and the context are the models that can be used for code generation (models at design time) and in models at runtime.

1.5 Overview of the Work

Figure 1.2 shows an overview of the work performed as part of this dissertation. It is structured in five different rows: row 1 identifies the challenge that is addressed; row 2 shows the research questions about the challenge; row 3 lists the scientific publications generated; row 4 lists the research projects where the work has been contributed to; row 5 lists the industrial partners where the solutions have been evaluated.

Challenge	Dynamic Feature Localization in Models	Bug Localization in Models	
Research Questions	**RQ1:** Using information from runtime traces	**RQ2:** Applicability of source code techniques	**RQ3:** Systems with dynamic reconfigurations
Publications	MRT '15, MRT '16, SAM '16, ECMFA '17	ISD '17, MoDELS '18, SoSyM '19	
	SANER '16		
Funded research projects	VARIAMOS: Model-Driven Variability Extraction for Software Product Line Adoption; Spanish National R+D+i Plan and ERDF funds - TIN2015-64397-R		
	REVaMP^2 : Round-trip Engineering and Variability Management Platform and Process; Information Technology for European Advancement - ITEA 3 Call 2		
Industrial partners	BSH: Home Appliances Group; Induction hob firmware variability extraction and management tool		
	CAF: Variability modeling, code generation and evolution for railway system software		

Figure 1.2: Overview of our contributions

For the first challenge (Dynamic Feature Localization in Models), one research question is identified (RQ1) and four publications are presented in chronological order (MRT'15 [7], MRT'16 [8], SAM'16 [9], and ECMFA'17 [10]). For the second challenge (Bug Localization in Models), two research questions are identified (RQ2, and RQ3) and three publications are presented in chronological order (ISD'17 [11], MoDELS'18 [13], and SoSyM'19 [12]). In addition, there is a transverse publication (SANER'16 [15]), which was the base for detecting these challenges.

The work presented in this dissertation has made contributions two projects: VARIAMOS, a Spanish national research project whose objective is the extraction of variability in the form of model fragments to achieve the adoption of SPL approaches; and REVaMP2, an international ITEA 3 Call 2 project whose main objective is the creation of a holistic platform and process for variability extraction and management over time.

The work presented in this dissertation was evaluated with the collaboration of two industrial partners: BSH, the leading manufacturer of home appliances in Europe with whom we have collaborated in the creation of a variability extraction and management tool for induction hob firmware; and CAF, a worldwide provider of railway solutions with whom we have collaborated in the creation of a solution for managing the variability of the existing software in the railway systems.

1.6 Structure of the dissertation

This dissertation is divided into five parts:

Part I presents the introduction and some background and then discusses the state of the art.

1 Introduction This section introduces the motivation for the dissertation, the challenges that are addressed, the contribution, the overview of the work done, the methodology followed and the structure of the dissertation.

2 Background This section presents some background related to the topics covered in the dissertation. Specifically, we present Model Driven Engineering (MDE), Models at runtime, and Information Retrieval.

3 State of the Art This section discusses the state of the art in relation to the scope of this dissertation (feature localization at runtime, bug localization at design time and bug localization at run time) and motivates the solutions presented.

Part II focuses on Feature and Bug Localization in Models.

4 Overview of the approaches This chapter presents the outline of the dissertation, the running example used to illustrate the approaches, and some feature and bug examples.

5 Dynamic Feature Localization in Models This chapter presents the dynamic feature localization approach at the model trace level and at the model level.

6 Bug Localization in Models with an Evolutionary Algorithm This chapter presents the bug localization approach in design time models.

7 Evolutionary Algorithm for Bug Localization in the Reconfigurations of Models at Runtime This chapter presents our bug localization approach in reconfigurations for systems with models at runtime.

Part III focuses on case studies and evaluation.

8 Case Studies This chapter introduces the domains in which we have evaluated our approaches.

9 Evaluations This chapter presents the details of the evaluations performed to validate our approaches, the measurements and the statistical analysis used, and the results obtained.

Part IV presents the discussion and conclusion.

10 Discussion This chapter includes the lessons learned, the issues and find-

ings detected with our approaches, and some comparisons to other works.

11 Conclusion This chapter includes the conclusion, the recapitulation of the research questions presented and their answers, the next steps in the research, and the related publications.

Part V includes the eight papers selected for the dissertation.

12 Dynamic Feature Localization in Models This chapter includes the five papers published in relation to the challenge of dynamic feature localization in models.

13 Bug Localization in Models This chapter includes the three papers published in relation to the challenge of bug localization in models.

2

BACKGROUND

Contents

2.1 Overview **14**

2.2 Model Driven Engineering **14**

2.2.1 Domain Specific Languages 15

2.3 Models at Runtime **16**

2.3.1 Autonomic computing and reconfigurations 17

2.4 Information Retrieval **18**

2.1 Overview

In this chapter the background of the dissertation is introduced. The background in this case is conformed by the approaches that are related to the objectives of this work. Therefore, this chapter provides a basic background for understanding the overall dissertation work. Specifically, we present Model Driven Engineering (MDE), Models at runtime, and Information Retrieval.

First, we present Model Driven Engineering, which is a software development methodology that focuses on creating and exploiting domain models, which are conceptual models of all the topics related to a specific problem.

Second, we present Models at runtime, which is a paradigm that seeks to extend the applicability of models produced in model driven engineering approaches to the runtime environment.

2.2 Model Driven Engineering

Model-driven engineering (MDE) is a software development methodology that focuses on creating and exploiting domain models, which are conceptual models of all the topics related to a specific problem.

The MDD methodology aims to increase productivity by maximizing compatibility between systems, simplifying the design process (through models of design patterns that are repeated in the application domain), and promoting communication between individuals and teams working in the system (through standardization of terminology and best practices in the application domain).

One of the best-known MDD initiatives is the model-driven architecture (MDA) of the Object Management Group (OMG) [16]. MDA is a specific realization of the Model Driven Development (MDD) [17] paradigm. Both MDD and MDA are within the MDE methodology [18].

The term Model Driven Engineering was introduced by Kent in [18] and it has been widely used in the literature for referring a more general approach to MDD than the one proposed by MDA. Models and model transformations are also at the center of this approach. As stated in [19], "the strong interest in models relies in the hope of being able to bridge the gap between words and codes". In words of Fondement [20], MDE "attempts to organize new efforts in these directions by proposing a framework (1) to clearly define methodologies, (2) to develop systems at any level of abstraction, and (3) to organize and automate the testing

and validation activities". In general, there is not any specific characteristic which differentiates the approaches referred as MDE and as MDD.

But the arrival of the MDD and MDA changed the way of using models in the development of software. As stated by Agrawal [21]: "the models are not merely artifacts of documentation, but 'living documents' that are transformed into implementations. This view radically extends the current prevailing practice of using UML: UML is used for capturing some of the relevant aspects of the software, and some of the code (or its skeleton) is automatically generated, but the main bulk of the implementation is developed by hand. MDA, on the other hand, advocates the full application of models, in the entire life-cycle of the software product."

The rise of all these methodologies suggest that this could be the right way to increase the productivity and the quality in the software development area.

2.2.1 Domain Specific Languages

Domain specific languages play key role in several of the MDD approaches. According to [22], a domain specific language (DSL) is "a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain."

DSLs are not a new topic, but the current stress on MDD have focused the interest of both academy and industry on this kind of languages. As stated by [22], the older programming languages all came into existence as dedicated languages for solving problems in a certain area. DSLs are tightly related to the Domain Engineering field. In words of Tolvanen [23], the main focus of Domain Engineering is finding and extracting domain terminology, architecture and components. It is important to note that two point of view when dealing with the domain concept can be considered [24]:

Conceptual domain. From this point of view, a domain is a set of interrelated real-world concepts. For instance, the *health-care* domain contains concepts like *"medical center"*, *"patient"*, *"disease"*, *"medicament"*, etc. As another example, the *industrial factory* domain contains concepts like *"product line"*, *"stock"*, *"worker"*, etc.

Systems domain. From this point of view, *"a domain is characterized by a set of systems that share some common defining features"* [24]. These systems

usually address a common problem area and conceivably share a common solution structure. In this case, we can talk about the expert systems domain, the database-based systems domain, the control/monitoring systems domain, the software games domain, etc.

Many benefits due to the use of DSLs can be found in the literature. For instance, according to [22]:

- DSLs allow solutions to be expressed in the idiom and at the level of abstraction of the problem domain. Consequently, domain experts themselves can understand, validate, modify, and often even develop DSL programs.
- DSL programs are concise, self-documenting to a large extent, and can be reused for different purposes.
- DSLs enhance productivity, reliability, maintainability, and portability.
- DSLs embody domain knowledge, and thus enable the conservation and reuse of this knowledge.
- DSLs allow validation and optimization at the domain level.

But some drawbacks have been also identified. These drawbacks are related to the associated cost (for designing, implementing, and learning the DSL) and the specific nature of the language (possible lack of expressiveness and/or loss of efficiency).

Some researchers think that *"the success of visual notations as commonly used domain specific languages is contingent on making similar tools and concepts for visual languages a commodity that can be readily used and understood by a wide audience, effectively lowering the initial hurdle to adoption* [25]. Hopefully, the number and quality of tools for implementing DSLs is growing and, therefore, a widely use of DSLs could be foreseen.

2.3 Models at Runtime

In MDE, a model is an abstraction or reduced representation of a system that is built for specific purposes. The models at runtime community shares this view of what constitutes a model and seeks an understanding of the roles that such models can play at runtime [26].

The models should represent the system and should be linked in such a way that they constantly mirror the system and its current state and behavior. If the system changes, the representations of the system, the models, should also change, and vice versa. It is critical that such representations be causally connected. This is an important requirement for adaptive systems for two reasons:

- the model as interrogated should provide up-to-date and exact information about the system to drive subsequent adaptation decisions; and
- if the model is causally connected, then adaptations can be made at the model level rather than at the system level.

Runtime models can support adaptation decisions by humans through adaptation agents embedded in the system itself or through combinations of both. The trend is to support increasing automation of decision making with respect to adaptation of systems as captured by the impulse towards autonomic computing, wherein agents may learn appropriate strategies for effective systems management. Models at runtime offers and important contribution to the field of autonomic computing providing metainformation to drive autonomic decision making [26].

2.3.1 Autonomic computing and reconfigurations

Autonomic computing has evolved as a discipline to create software systems and applications that self-manage in a bid to overcome the complexities and inability to maintain current and emerging systems effectively. To this end autonomic endeavors cover the broad span of computing from end-to-end applications to infrastructure middlewares and are already demonstrating their feasibility and value.

To achieve autonomic computing, IBM suggested a reference model for autonomic control loops [27], which is sometimes called the MAPE-K (Monitor, Analyze, Plan, Execute, Knowledge) loop. This model is being used more and more to communicate the architectural aspects of autonomic systems. The MAPE-K autonomic loop is similar to, and probably inspired by, the generic agent model proposed by Russel and Norvig [28], in which an intelligent agent perceives its environment through sensors and uses these percepts to determine actions to execute on the environment.

In the MAPE-K autonomic loop, the managed element represents any software (i.e. models at runtime) or hardware resource that is given autonomic behavior

by coupling it with an autonomic manager. Actuators carry out changes to the managed element.

The data collected by the sensors allows the autonomic manager to monitor the managed element and execute changes through actuators. The autonomic manager is a software component that ideally can be configured by human administrators using high-level goals and uses the monitored data from sensors and internal knowledge of the system to plan and execute, based on these high-level goals, the low-level actions that are necessary to achieve these goals. The internal knowledge of the system is often an architectural model of the managed element, and the goals are usually expressed using Event Condition Action rules [29].

2.4 Information Retrieval

Information Retrieval (IR) is the task, given a set of documents and a user query, of finding the relevant documents. There are many information retrieval techniques: program analysis dependencies [30, 31, 32, 33], textual similarity [34, 35, 36, 37], trace analysis [38, 39, 40, 41], type systems [42, 43, 44, 45] or propositional logic [46, 47, 48, 49]. In this work, we focus on information retrieval techniques based on textual similarity.

The IR techniques based on textual similarity, are based on mathematical and statistical methods to determine the similarity between different collections of texts. Three of the most popular IR techniques are: Vector Space Model (VSM) [50], Latent Semantic Indexing (LSI) [51] or Latent Dirichlet Allocation (LDA) [52].

The Vector Space Model (VSM) [50] is an algebraic model, in which a document D is represented as an m-dimensional vector, where each dimension corresponds to a distinct term and m is the total number of terms used in the collection of documents. The document vector is written as, where w_i is the weight of term t_i that indicates its importance. If document D does not contain term T_i then weight w_i is zero.

The Latent Semantic Indexing (LSI) [51] takes into account the number of occurrences of a series of keywords or 'queries' in long texts or 'documents'. As a result, LSI can be used to obtain measures of similarity between the names (or descriptions) of the features and the source code that implements them.

The Latent Dirichlet Allocation (LDA) [52] is a 'generative probabilistic model' of a collection of composites (documents) made up of parts (words and/or phrases).

The probabilistic topic model estimated by LDA consists of two tables (matrices). The first table describes the probability or chance of selecting a particular part when sampling a particular topic (category). The second table describes the chance of selecting a particular topic when sampling a particular document or composite.

The current results are ambiguous and contradictory about which technique provides the best performance [53]. However, some works state that LSI performs better working with bug reports [54] or with text [55], while VSM works better with source code. This is due to the reason that VSM works very well in case of exact match while LSI retrieves relevant documents based on the semantic similarity.

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

3

STATE OF THE ART

Contents

3.1 Overview **22**

3.2 Feature Localization at Runtime **23**

3.2.1 Motivation of our dynamic feature localization on model approaches 25

3.3 Bug Localization at Design time **25**

3.3.1 Motivation of our bug localization on design time model approach 28

3.4 Bug Localization at Runtime **29**

3.4.1 Motivation of our bug localization on models at runtime approach 31

3.1 Overview

This chapter presents the state of the art related to this dissertation. This is divided taking into account the two main challenges: dynamic feature localization in models and bug localization in models. Both challenges are highly related, both are common tasks in software maintenance, and both pursue the fact to find a target artifact related to other artifacts. Although the solutions can be interchangeable, each of them has been developed focusing on one of the challenges. In other words, you can use a feature localization approach to locate bugs. However, this approach may not take into account special properties for bug localization that are not necessary for feature localization.

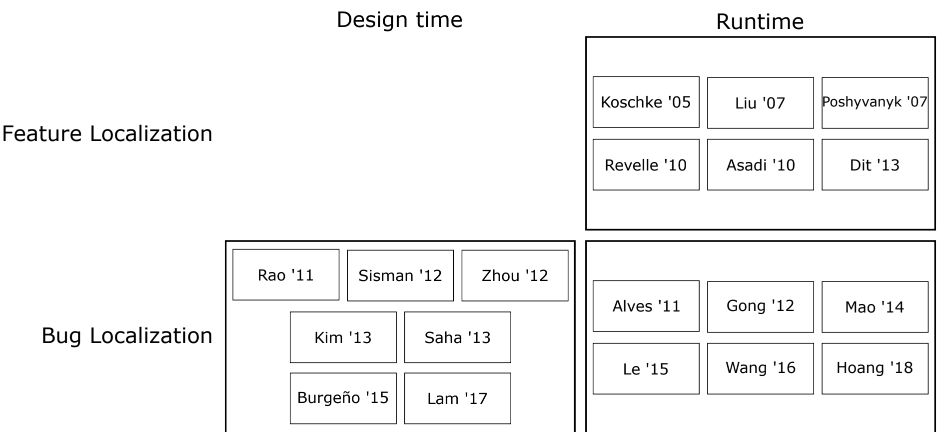


Figure 3.1: Overview of the works related to this dissertation

Figure 3.1 shows an overview of the related works. These works are presented in the next sections separated by the challenges. In addition, all of the works are going to be described as follows:

Identifier: The name of the work that appears in Figure 3.1.
Technique: The technique or techniques used to locate features or bugs.
Artifact: The type of the artifacts used as input, they can be code or models.
Program code □, Models □
Input: The artifacts that have to be provided to perform the localization.
Output: The results and how are they outputted by the technique used.
Application: The domain in which they evaluate their technique.

3.2 Feature Localization at Runtime

There are many research efforts in dynamic feature location techniques. Most of these works are based on program-code analysis. Usually, this kind of works combines several techniques for feature location.

Kosche and Quante [33] developed a semi-automatic technique that combines static and dynamic analysis techniques. They use formal concept analysis (FCA) to explore the results of the dynamic analysis.

Identifier: Koschke '05
Technique: Dynamic and formal concept analysis.
Artifact: Program code ☑ , Models ☐
Input: Execution trace from a scenario.
Output: List of basic blocks executed and the number of executions of each one.
Application: Two compilers: sdcc and cc1.

Liu et al. [34] combine information from an execution trace and from the comments and identifiers from the source code. They executed a single scenario which executes the desired feature. All the executed methods are identified based on the collected trace using Latent Semantic Indexing (LSI). The result is a ranked list of executed methods based on their textual similarity to a query.

Identifier: Liu '07
Technique: Dynamic analysis and information retrieval.
Artifact: Program code ☑ , Models ☐
Input: Developer query and an execution trace from a scenario.
Output: Ranking of methods that appear in the execution trace.
Application: jEdit and Eclipse.

Poshyvanyk et al. [41] introduced the probabilistic ranking of methods. Their approach is based on execution scenarios and information retrieval. The result is a rank of program elements, their approach can varied the parameters to give more weight to the probabilistic ranking or to information retrieval.

Identifier: Poshyvanyk '07
Technique: Probabilistic ranking and information retrieval.
Artifact: Program code ☑ , Models ☐
Input: Developer query and an execution trace from a scenario.

Output: Ranking of program element.
Application: Eclipse and Mozilla.

Revelle et al. [36] apply data fusion for feature location. Their technique combines information from textual, dynamic, and web mining analysis applied to a software system. Their input is a single scenario that executes the feature; after running the scenario, they constructed a call graph that contains only the methods that were executed. Then, they apply a web-mining algorithm, and the system filters out low-ranked methods. The remaining set of methods is scored using LSI based on their relevance to the input query that describes de feature.

Identifier: Revelle '10
Technique: Dynamic and web mining analysis, and information retrieval.
Artifact: Program code ☑ , Models ☐
Input: Developer query and an execution trace from a scenario.
Output: Ranking of executed methods.
Application: Two java systems: Eclipse and Rhino.

Asadi at al. [56] proposed a feature location technique that identifiers cohesive and decoupled fragments from execution traces which are related to concepts. The trace is preprocessed to remove irrelevant methods and to compress the trace. Then, the trace is used as an input for a genetic algorithm. The fitness function used is based on a conceptual cohesion metric. The result is a set of fragments that contain methods that are highly cohesive, and which are highly decoupled with other fragments.

Identifier: Asadi '10
Technique: Dynamic analysis and genetic algorithm.
Artifact: Program code ☑ , Models ☐
Input: Execution trace from a scenario.
Output: Fragments of methods.
Application: Two java systems: ArgoUML and JHotDraw.

Dit et al. [57] present a data fusion model for feature location that is based on the idea that combining data from several sources in the right proportions will be effective at identifying a feature. The data fusion model defines different types of information that can be integrated to perform feature location including textual, execution, and dependence. Textual information is analyzed by information retrieval, execution information is collected by dynamic analysis, and dependencies are analyzed using link analysis algorithms.

Identifier: Dit ’11
Technique: Dynamic analysis, information retrieval, and link analysis algorithms.
Artifact: Program code ☑ , Models ☐
Input: Developer query and an execution trace from a scenario.
Output: Ranking of methods.
Application: Three java systems: Eclipse, Rhino and jEdit.

3.2.1 Motivation of our dynamic feature localization on model approaches

Similarly to our techniques, all of the dynamic feature location approaches presented above use information from different sources. First, they delimit the search space by applying some dynamic analysis technique. Thus, they take advantage of the runtime information, in the same way that we do. Then, they present their results in the form of sets or rankings of methods or program elements.

However, we have detected that none of the presented approaches take advantage of the use of models at runtime. In fact, a recent survey [58] reveals that none of the approaches under their study leverage models at runtime for feature (or bug) localization. Models at runtime correspond to the models that contain knowledge about the environment and the system itself. They support learning of that knowledge [59]. Hence, models at runtime can be a great source of information since they can be traversed and consulted to provide up-to-date information.

In addition, the language used to define the models is closer to the natural language than the program code. Then, the information retrieval techniques used in the presented works could obtain more accurate results applied to models. To address the dynamic feature location in systems with models at runtime, we propose the adaptation of program-code feature location techniques to work with models at runtime. Specifically, we use dynamic feature location to extract execution model traces and information retrieval to filter the relevant model elements for the desired feature. As a result, the user obtain a ranked list of model elements related to the feature to be located.

3.3 Bug Localization at Design time

In recent years, many bug localization approaches have been proposed [5]. These approaches are usually IR-based approaches, and some of them add the defect lo-

calization principle. Since our bug localization approach applies these techniques, in this section, we review some relevant works in the literature.

Rao and Kak [60] apply information retrieval for bug localization. They compare five basic IR models, and some variants thereof, for retrieval from software libraries for the purpose of bug localization.

Identifier: Rao '11
Technique: Information retrieval.
Artifact: Program code ☑ , Models ☐
Input: Bugs with textual descriptions, names of patch files that correspond to the bugs, and the source files that form the entire library.
Output: Ranking of retrieved files.
Application: iBUGS, a benchmarked bug localization dataset.

Sisman and Kak [61] include the defect localization principle in their approach. They utilize time decay in weighting the files in a probabilistic information retrieval model.

Identifier: Sisman '12
Technique: Information retrieval and defect localization principle.
Artifact: Program code ☑ , Models ☐
Input: Set of reported bugs, the collection of the files relevant for these bugs.
Output: Ranking of retrieved files.
Application: iBugs dataset for AspectJ.

Zhou [62] perform bug localization applying a information retrieval technique. They propose BugLocator. Their approach uses an initial bug report to rank program code files in descending order based on their relevance to the bug report.

Identifier: Zhou '12
Technique: Information retrieval.
Artifact: Program code ☑ , Models ☐
Input: Set of reported bugs, the collection of the files relevant for these bugs.
Output: Ranking of retrieved files.
Application: Eclipse, AspectJ, SWT and ZXing.

Kim et al. [63] propose a one-phase and two-phase prediction model to recommend files to fix. In the one-phase, they create features from textual information

and metadata of bug reports, apply Nave Bayes to train the model using previously fixed files as classification labels, and then use the trained model to assign multiple source files to a bug report. In the two phase, they first apply their one-phase model to classify a new bug report as either 'predictable' or 'deficient' and then make predictions only for 'predictable' reports.

Identifier: Kim '13
Technique: Machine learning and information retrieval.
Artifact: Program code ☑ , Models ☐
Input: Set of reported bugs, the collection of the files relevant for these bugs.
Output: Set of files to fix where each file is associated with a probability of being the file to fix.
Application: Mozilla 'Firefox' and 'Core' packages.

Saha et al. [64] introduce an automatic bug localization tool based on the concept of structured information retrieval. They extract and model code constructs like structured documents, and we show how a seemingly trivial change to how camel case identifiers are indexed yields significantly improved localization accuracy.

Identifier: Saha '13
Technique: Information retrieval.
Artifact: Program code ☑ , Models ☐
Input: Set of reported bugs, the collection of the files relevant for these bugs.
Output: List of ranked files.
Application: Eclipse, AspectJ, SWT, and ZXing.

Burgueño et al. [65] present a static approach to trace errors in model transformations. Taking as input elements an ATL model transformation and a set of constraints that specify its expected behavior. Their approach automatically extracts the footprints of both artifacts and compares transformation rules and constraints one by one, obtaining the overlap of common footprints. The output is three matching tables that can be used by the software engineer to trace the rules that can be the cause of broken constraints due to faulty behavior.

Identifier: Burgueño '15
Technique: Matching tables using meta-model footprint of the constraints and the transformation rules.

Artifact: Program code ☐, Models ☑
Input: Set of model transformation and constraints.
Output: List of rules of a model transformation that may be the cause of a faulty behavior.
Application: UML2ER, CPL2SPL, BT2DB, and Ecore2Maude.

Lam et al. [66] present an approach that uses deep neural network (DNN) in combination with an information retrieval technique, rVSM. rVSM collects the feature based on the textual similarity between bug reports and source files. DNN is used to learn ro relate the terms in bug reports to potentially different code tokens and terms in source files.

Identifier: Lam '17
Technique: Deep neural network and information retrieval.
Artifact: Program code ☑ , Models ☐
Input: Set of reported bugs, the collection of the files relevant for these bugs.
Output: Top-ranked files.
Application: AspectJ, Birt, Eclipse UI, JDT, SWT, and Tomcat.

3.3.1 Motivation of our bug localization on design time model approach

Similar to the previous section, all of these bug localization approaches use information from different sources. In the same way, most of them combine different techniques to locate bugs, as we do. Then, their output is a ranking of files relevant for the bug.

Only one of the approaches presented takes into account models, specifically, model transformation as the source of the bug. The res of the above works present approaches that only take into account the source code as the artifact that represents the bug. A recent survey [5] reveals that none of the bug localization approaches takes into account models as the source of the bugs. When models are used for code generation, addressing bugs at the model level must not be neglected.

We developed an approach that applies similar techniques (defect localization principle and information retrieval) on models. Specifically, we use a multi-objective evolutionary algorithm that uses both the similarity to the bug report and the timespan weighting as fitness functions. As a result, the user obtains a ranked

list of model fragments. This list can be ordered following the similarity to the bug report or the most recent model fragment modifications.

Our approach and the approach presented in [65] are complementary. If we use a model with a bug to generate program code, the bug will be transferred to the code. In the same way, if we use a transformation rule with a bug to generate program code, the bug will be transferred to the code. Hence, when we work with models for generating program code, it is important to ensure that there is no bugs in the model or in the transformations that will generate the program code.

3.4 Bug Localization at Runtime

With the massive size and scale of software systems today, traditional fault localization techniques are not effective in isolating the root causes of bugs [5]. Sometimes, with design time approaches, we include extra information not relevant for a bug that occurs at runtime. To exclude this extra information, we need approaches that take into account the runtime behavior. Some of these approaches are the following.

Alves et al. [67] combine dynamic slicing and spectrum-based techniques. They rank all of the statements in a program based on their level of suspiciousness, which is calculated by using a spectrum-based technique (the Tarantula technique). Then, they generate a dynamic slice with respect to a failure-indicating variable at the failure point. The statements that are not in the slice are removed from the ranking to reduce the search domain.

Identifier: Alves '11
Technique: Dynamic slicing and spectrum-based.
Artifact: Program code ☑ , Models ☐
Input: Test suite coverage information.
Output: Covered statements ranked in order of suspiciousness.
Application: 50 subjects obtained from 2 applications from the Software-artifact Infrastructure Repository.

Gong et al. [68] propose an interactive localization technique called TALK. This approach incorporates programmers' feedback into spectrum-based fault localization techniques. When the programmer receives the ranking of program elements that can cause the bug, he or she can judge the correctness of each element and provide this information as feedback to reorder the ranking.

Identifier: Gong '12
Technique: Incorporates user feedback to spectrum-based fault localization.
Artifact: Program code ☑ , Models ☐
Input: Program spectra.
Output: Ranked list of suspicious program elements.
Application: Five real C programs and seven Siemens test programs from the Software-artifact Infrastructure Repository.

Mao et al. [69] use dynamic slicing and statistical bug localization. They utilize program slices of a set of test runs to capture the influence of a program entity's execution on the output, and they use statistical analysis to measure the level of suspiciousness of each program entity being faulty. Their approach is called approximate dynamic backward slice.

Identifier: Mao '14
Technique: Dynamic slicing and statistical bug localization.
Artifact: Program code ☑ , Models ☐
Input: Program slices (a particular execution of a program in a specific input).
Output: A ranking list of all statements in descending order of their suspiciousness.
Application: Two standard benchmarks from the Siemens suite and space, and three UNIX utility programs.

Le et al. [70] and Hoang et al. [71] combine information retrieval and spectrum-based techniques. In [70, 71], they present two approaches that utilize multimodal information from both bug reports and program spectra to localize bugs.

Identifier: Le '15 and Hoang '18
Technique: Information retrieval and spectrum-based.
Artifact: Program code ☑ , Models ☐
Input: Bug reports and program spectra.
Output: Ranked list of methods.
Application: AspectJ, Ant, Lucene, Rhino, Lang, Math, and Time.

Wang and Lo [72] include the Defect Localization Principle in their approach. They present AmaLgam+, which is a method for locating relevant buggy files that puts together fives sources of information: version history, similar reports, structure, stack traces, and reporter information.

Identifier: Wang ’16
Technique: Bug prediction, information retrieval, structured retrieval for bug localization, and reporter’s information.
Artifact: Program code ☑ , Models ☐
Input: A bug report to be localized, a set of source code files of the system, a history of commits made to the system, and a set of older bug reports stored in a bug tracking system.
Output: Ranking of files.
Application: AspectJ, Eclipse, SWT, and ZXing.

3.4.1 Motivation of our bug localization on models at runtime approach

All of the above works take advantage of execution information to perform bug localization. They combine several known techniques for bug localization while our approach uses information retrieval as the fitness function for an evolutionary algorithm.

In addition, as in the previous section, none of the approaches takes into account the models or their reconfigurations at runtime as the source of the bugs. In many systems with models at runtime, the model experience reconfigurations at runtime due to context changes being these reconfigurations a source of bugs. A recent Search-based Model-driven Engineering survey [73] reveals that none of the bug localization approaches take into account the bugs caused by the reconfigurations of a models at runtime system.

Our approach is focused on locating bugs that appear as the result of dynamic reconfigurations of the system due to context changes. It is an evolutionary algorithm guided by a fitness function that considers the similarity to the description of the bug report. The output is a ranked list of reconfiguration sequences intended to identify the reconfiguration rules that are relevant to the bug.

Part II

Feature and Bug Localization in Models

4

OVERVIEW OF THE APPROACHES

Contents

4.1 Overview **36**

4.2 Outline for this dissertation **36**

4.3 Running example **38**

4.4 Feature and bug examples **40**

4.4.1 Feature example 40

4.4.2 Bug example 41

4.4.3 Bug in reconfigurations example 41

4.1 Overview

This chapter presents the overview of the approaches that are going to be presented in the following chapters. We identify the main elements of each of the approaches and present an outline that we will use for explaining each of them.

In addition, this chapter presents the running example that will be used to illustrate the approaches. Specifically, we present our running example extracted from one of our industrial partners, BSH.

Finally, this chapter also presents three examples: (1) a feature example, (2) a bug example, and (3) a bug in reconfigurations example. All of them are illustrated using the BSH domain.

4.2 Outline for this dissertation

Table 4.1 shows the outline for this dissertation. The work done has been developed in model-based systems, and it is divided in two topics: Feature localization and Bug Localization. The rest of the columns are explained in the following paragraphs.

Approach. The approach column shows the name of the approaches. Each of the approaches proposes a process that try to solve one of the challenges. The process followed is different in each case. Figure 4.1 shows the relationship

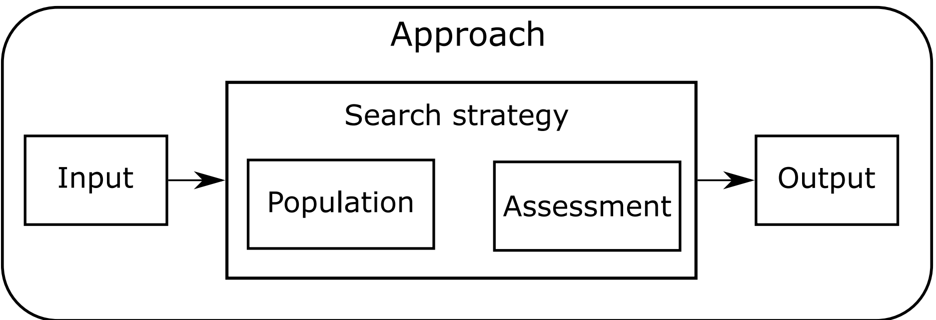


Figure 4.1: Overview of each of the approaches

among all the elements that appear in the columns of the Table 4.1. The input, the population, the search strategy, the assessment and the output are specific for each of the approaches. Hence, we can define each of the approaches by specifying each of these elements.

Input. The input column shows the artifacts that have to be provided to the approach to perform the localization. The artifacts are where the feature or bug will be located, and the information that has to be provided about the feature or the bug that is going to be located.

The domain experts have to identify and provide the artifact that contains the feature, or the bug being located. In addition, they have to provide the information that will be used to guide the approach. Some of this information will vary depending on the type of artifacts where the feature of the bug will be located and the information available. Therefore, depending on the

	Feature localization	**Bug localization**	
Approach	**DFL:** Dynamic Feature Localization	**BLiMEA:** Bug Localization in Models with an Evolutionary Algorithm	**EBRo:** Evolutionary algorithm for Bug localization in the Reconfigurations of models at runtime
Input	- Models - Feature description	- Models - Bug report	- Reconfiguration rules - Initial model - Bug report
Population	- Models - Model elements	- Model fragments	- Reconfiguration sequences
Search Strategy	- Dynamic analysis	- Multi-objective evolutionary algorithm	- Evolutionary algorithm
Assessment	- Information retrieval	- Information retrieval - Defect localization principle	- Information retrieval
Output	- Ranking of model elements with values above threshold	- Ranking of model fragments	- Ranking of reconfiguration sequences

Table 4.1: Outline of the dissertation

information available, the experts will select different sources of information.

Population. The population column shows the form of the individuals of the population. Each individual of the population is a candidate for the solution. Therefore, as each individual of the population can be a solution, we can see the population as the search space in which we are going to locate the feature or the bug.

Search strategy. The search strategy column shows the technique used to explore the search space. In the majority of the industrial domains, the search space can be huge. For example, in a model with 100 elements, there are 10^{29} candidates for a feature realization.

As the search space can be very large, we use the search strategy in order to allows us to obtain a smaller search space or to help us to explore it in an optimal way.

Assessment. The assessment column shows the heuristic used to evaluate each individual of the population. To do so, the approach assigns a value to each of the solution candidates based on their quality as the artifact that contains the feature of the bug.

Output. The output column shows the artifacts that are shown to the user of the approach. As there are not a perfect approach that obtains the perfect solution for the feature of the bug that we want to locate, the approaches provide the results in the form of rankings of possible solutions.

In the next part of this dissertation, we are going to present the contributions by means of the approaches of the Table 4.1, following each row of the table and specifying each one of the elements of the columns.

4.3 Running example

This section presents the Induction Hobs Domain, including the Domain Specific Language used by our industrial partner, BSH, to specify their product models. The language and graphical representations presented in this section will serve as the basis of the running example used to illustrate the rest of the dissertation.

The newest induction hobs (IHs) feature full cooking surfaces, where dynamic heating areas are automatically generated and activated or deactivated depending on the shape, size, and position of the cookware placed on the top. In addition, there has been an increase in the type of feedback provided to the user while cooking. All of these changes have been made possible at the expense of increasing the complexity of the software behind IHs.

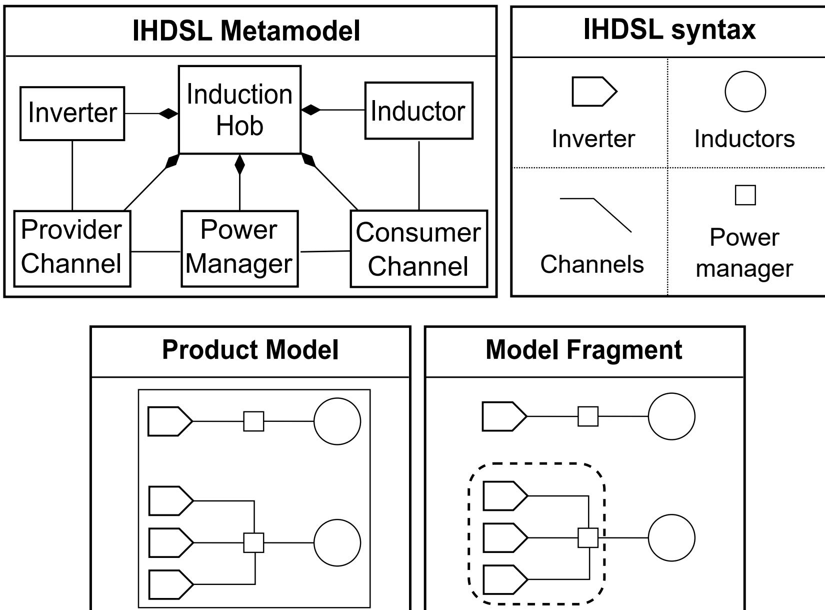


Figure 4.2: IHDSL metamodel, syntax, product model, and model fragment realization

The domain-specific language used by our industrial partner to specify the induction hobs (IHDSL) is composed of 46 meta-classes, 47 references among them, and more than 180 properties. To gain legibility and due to intellectual property right concerns, in this section, we show a simplified subset of the IHDSL (see Figure 4.2, IHDSL metamodel and IHDSL syntax). However, the evaluations were performed using the full IHDSL that is used in BSH.

Inverters are in charge of transforming the input electric supply to match the specific requirements of the IH. Then, the energy is transferred to the inductors through the channels. There can be several alternative channels, which enable different heating strategies depending on the cookware placed on top of the IH at runtime. 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.

The product model in Figure 4.2 depicts an example of a product model that is specified with the IHDSL. The product model contains four inverters that are used to power two different inductors. The upper inductor is powered by a single inverter, while the lower inductor is powered by the combination of three different inverters. Power managers act as hubs to perform the connection between the inverters and the inductors.

To formalize the solution of our approach as model fragments, we use common variability language (CVL) [74, 75], due to its capabilities to formalize a set of model elements as a model fragment. The model fragment in Figure 4.2 shows an example of a model fragment of the product model (the product model in Figure 4.2). The model fragment includes the three inverters (in charge of powering the lower inductor), the three channels, and the power manager that is used to aggregate and manage the power provided by those inverters. Then, the solution of our approach is formalized by means of CVL and shown to the engineers.

4.4 Feature and bug examples

This section presents some examples of feature and bugs in the Induction Hobs Domain.

4.4.1 Feature example

Figure 4.3 shows an example of a feature. The product models that appear on the left compose the family of induction hobs. In the right upper part appears the feature description that will be used by the approaches to locate the feature.

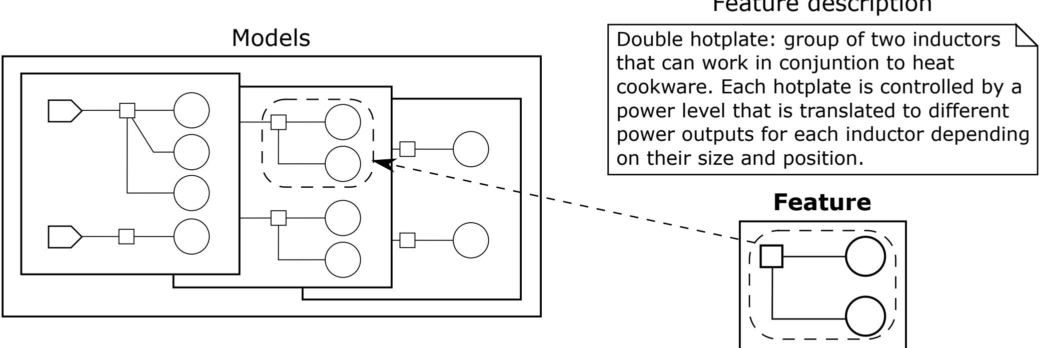


Figure 4.3: Example of a feature

The bottom right part shows the elements of the feature that corresponds to the feature description. These model elements form a model fragment. Hence, this model fragment is the one that realizes the feature.

4.4.2 Bug example

Figure 4.4 shows an example of a bug. The product model that appears on the left of the timeline is modified to make an improvement in the performance of its inductors. As a result, another product model is generated.

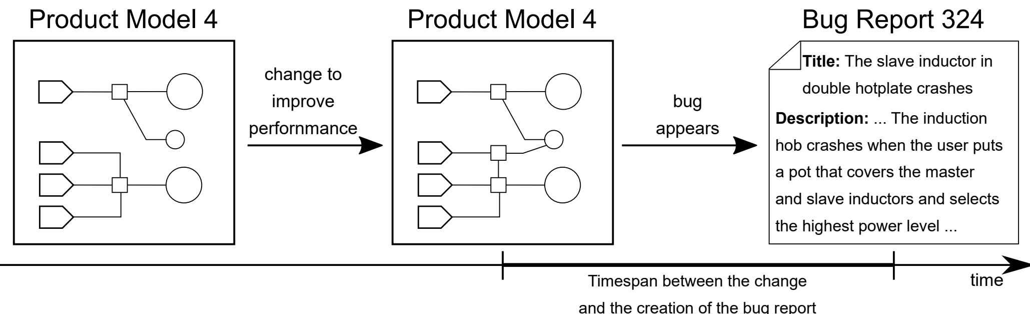


Figure 4.4: Example of a bug

This product model has a new power manager that connects the small inductor with the rest of the inverters. After some time of use, a bug appears, and a bug report is generated. In the reminder of this thesis, the timespan between the change and the creation of the bug report is called the modification timespan.

4.4.3 Bug in reconfigurations example

Figure 4.5 shows a reconfiguration that occurs in the induction hobs of BSH. In the initial configuration, the induction hob has two pots on top, heated through two inductors. The upper inductor is powered with one inverter and the bottom inductor is powered with three inverters.

When a bigger pot is placed in the upper inductor, the induction hob reconfigures itself. Another inductor is activated ('R1'), and more energy is needed to heat pot ('R2'). Therefore, the second inverter should give power to the upper inductors. However, the second inverter is not disconnected from the bottom power group (see the cross in Figure 4.5). This situation causes a bug, because when the user changes the power level of the upper inductors, the same power level will be applied to the bottom inductors, and vice versa.

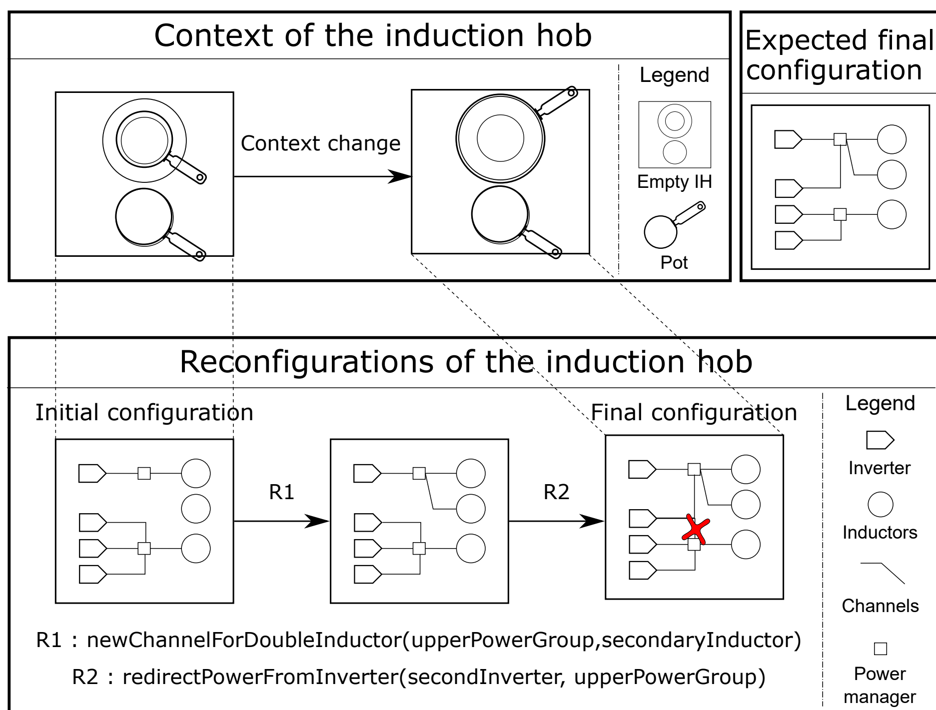


Figure 4.5: Example of a context change, a bugged reconfiguration, and the reconfiguration rules performed

This bug was solved by modifying the reconfiguration rule 'R2' so that in addition to redirecting the power of the inverter, the inverter is also disconnected from the previous power group (see expected final configuration in Figure 4.5).

5

DYNAMIC FEATURE LOCALIZATION IN MODELS

Contents

5.1 Overview **44**

5.2 Approach **44**

5.3 Input **45**

5.3.1 Model traces 45

5.4 Population **46**

5.5 Search Strategy **46**

5.6 Assessment: Information Retrieval **47**

5.6.1 Information Retrieval at model trace level 48

5.6.2 Information Retrieval at model level 49

5.7 Output **51**

5.1 Overview

This chapter presents DFL (see Figure 5.1), our dynamic feature localization approach for models at runtime [9, 10]. This approach applies Information Retrieval at model trace level and at model level. As a result, the approach generates a ranking with the most relevant model elements for feature to be located. Next sections describe the details of the approach.

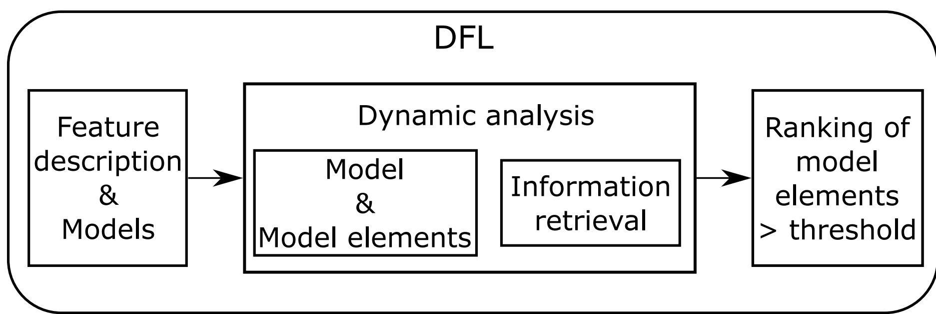


Figure 5.1: Overview of DFL

5.2 Approach

Figure 5.2 shows an overview of this approach. It is composed of three steps: dynamic analysis, information retrieval in the model trace, and information retrieval in a model from the model trace.

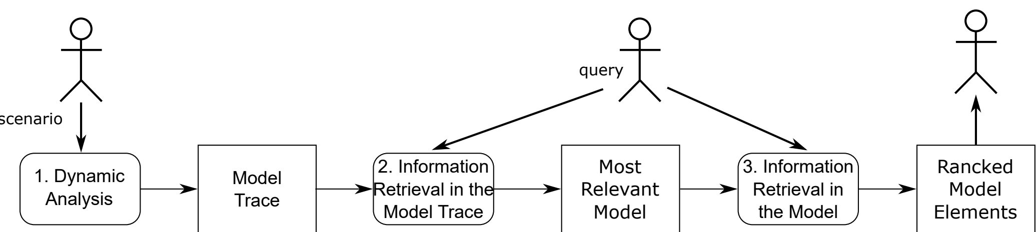


Figure 5.2: Overview of the Dynamic Feature Location Approach

In the first step, the software engineers execute a scenario, which involves the desired feature to be located. The execution information is recorded by a model trace of snapshots at runtime. Then, the model trace is used as input for the second step. Using information retrieval, the most relevant model for the target feature

is selected from the model trace. This model is used as input for the third step, which performs information retrieval at the model element level. As a result, the software engineers obtain a ranked list of model elements from the model, intended to identify the parts of the model that are significant for the target feature.

5.3 Input

In this approach we have three different inputs for each of the steps. For the first step, the input is the description of a scenario that will be executed. To describe the scenario the software engineers can use the description of the feature or their own knowledge about the domain.

The feature description of the target feature uses natural language. Typically, these descriptions can come from textual documentation of the products, comments in the code, bug reports or oral descriptions from the engineers. The knowledge of the engineers about the domain and the product models will be useful to select the textual description from the sources available.

For the second step, the input is the model trace extracted from the running scenario. Each trace is related to a set of snapshots of the runtime model. In this work, we compared two criteria to decide when a snapshot of the runtime model should be added to the trace: (1) configuration criterion, and (2) architecture criterion.

5.3.1 Model traces

In the configuration criterion, the snapshots are added to the trace when the runtime model corresponds to a target configuration of the system in a reconfiguration. That is, a snapshot is added when the system completes the changes from one configuration to another. In the architecture criterion, the snapshots are added to the trace when a change in the runtime model is performed. That is, a snapshot is added each time a component of the runtime model is deleted or created even if the model does not correspond to a target configuration of the system.

Figure 5.3 shows two different traces for the same reconfiguration. The upper part shows a trace composed by the configuration criterion. In the initial configuration, the induction hob has two pots on top, heated through two inductors. The upper inductor is powered with one inverter and the bottom inductor is powered with three inverters. When a bigger pot is placed in the upper inductor, the induction hob reconfigures itself. Another inductor is activated, and more energy is

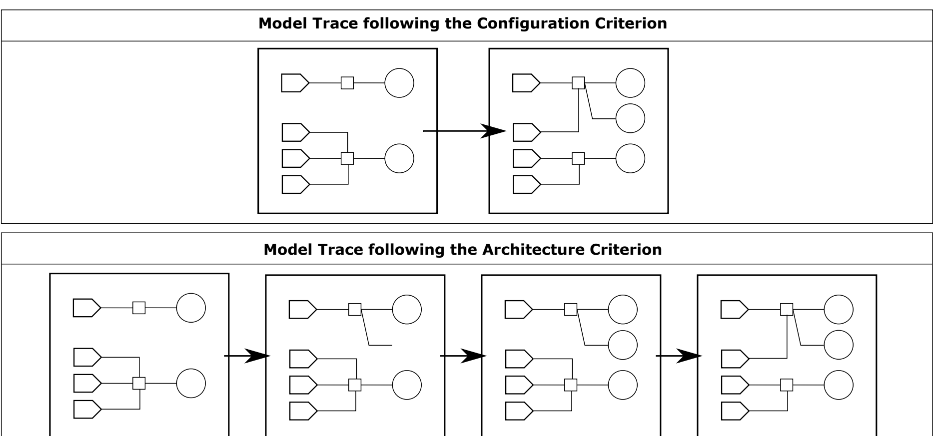


Figure 5.3: Different Model Traces following the different Criterion

needed to heat pot. Therefore, the second inverter should give power to the upper inductors. The bottom part shows a trace composed by the architecture criterion. The first snapshot and the last one are the same, as in the upper part of the figure. However, the rest of the snapshots give more detail on what actions were carried out in the reconfiguration from the first snapshot to the fourth one.

For the third step, the input is the model extracted from the model trace. This model is the one that is most relevant for the target feature.

5.4 Population

With this approach we have two types of individuals. For the second step, the population is composed by models that appear in the execution model trace. In this step, the goal is to obtain the most relevant model for the target feature.

For the third step, the population is composed by model elements from the most relevant model. In this step, the goal is to obtain the most relevant model elements for the target feature.

5.5 Search Strategy

As we presented, this approach is composed by three steps. In step 1: Dynamic Analysis is used to delimit the search space. Execution information is gathered via dynamic analysis, which is commonly used in program comprehension and involves executing a software system under specific conditions.

In our case, we design scenarios that run the target features in order to obtain model traces in which the target features are involved. In other words, executing the target feature during runtime generates a feature-specific execution trace.

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

5.6 Assessment: Information Retrieval

In step 2 and 3 we use Information Retrieval (IR) to asses each model in step 2 and to asses each model element in step 3. The information retrieval technique used is based on text. Textual information in source code (represented by identifier names and internal comments) embeds domain knowledge about a software system. In our case, textual information corresponds to the names, attributes and methods of the model elements. This information can be leveraged to locate the implementation of a feature through the use of IR. IR works by comparing a set of artifacts to a query and ranking these artifacts by their relevance to the query.

There are many popular IR techniques such as Vector Space Model (VSM) [76], Latent Semantic Indexing (LSI) [51] or Latent Dirichlet Allocation (LDA) [52]. The research findings are ambiguous and contradictory about which technique provides the best performance [53].

We apply Latent Semantic Indexing (LSI) to analyze the relationships between the description of the bug provided by the user and the model fragments. Besides that LSI provides good results when applied to source code [36, 34, 41], a recent work reveals that LSI performs better when applied to text [55]. Product models are representations at a higher abstraction level than the source code, and the language used to build them is closer to the bug description language; similar to text.

To perform LSI, our approach follows five main steps: creating a corpus, preprocessing, indexing, querying, and generating results:

Creating a corpus. In the first step of LSI, a document granularity needs to be chosen to form a corpus. A document lists all the text found in a contiguous section of source code (methods, classes, or packages). A corpus consists of a set of documents.

Preprocessing. Once the corpus is created, it is preprocessed. Preprocessing involves normalizing the text of the documents. For source code, operators

and programming language keywords are removed. In addition, identifiers and compound words are split.

Indexing. The corpus is used to create a *term-by-document matrix*. Each row of the matrix corresponds to each term in the corpus, and each column represents each document. Each cell of the matrix holds a measure of the weight or relevance of the term in the document. The weight is expressed as a simple count of the number of times that the term appears in the document. In other words, each term-document pair has a number that indicates the number of times this term appears as part of the names of attributes or methods of this model element.

Querying. A user formulates a query in natural language consisting of words or phrases that describe the feature to be located. Since LSI does not use a predefined grammar or vocabulary, users can originate queries in natural language.

Generating results. In LSI, the query and each document correspond to a vector. The cosine of the angle between the query vector and a document vector is used as the measure of the similarity of the document to the query. The closer the cosine is to 1, the more similar the document is to the query. A cosine similarity value is calculated between the query and each document, and then the documents are sorted by their similarity values.

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) [77]. SVD is a form of factor analysis, or more precisely, 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 that contains scaling values such that when the three components are matrix-multiplied, the original matrix is reconstructed.

5.6.1 Information Retrieval at model trace level

In step 2: Information Retrieval at model trace level, we use the model trace extracted in dynamic analysis. In addition, the software engineer has to formulate a query related to the feature that must be located. The model trace and the query can be leveraged to locate the most relevant model for the feature through the use of information retrieval.

For this approach, we adapt each step of the LSI technique to work with the model trace. The adaptation is performed as follows:

Creating a corpus. Each document corresponds to a model of the model trace extracted in the dynamic analysis. Each document (model) includes text from the names of the model elements and the names of the attributes and operations of the model elements that compose that model.

Preprocessing. The type of the attributes and the type of the parameters in the methods are removed. Then, all the identifiers are splitted. To do this, we apply Natural Language Processing (NLP) techniques, such as tokenizing, Parts-of-Speech (POS) tagging techniques, and stemming techniques [46, 78].

Indexing. In the term-by-document co-occurrence matrix, the terms (rows) correspond to the names of the model elements and the names of the attributes or operations of the model, and the documents (columns) correspond to the models that have appeared in the model trace. Each cell in the matrix contains the frequency with which the keyword of its row appears in the document denoted by its column.

Querying. We use the feature description to formulate the queries. Only the relevant terms are taken into account, and words such as determinants and connectors from the language are omitted. The query column represents the words that appear in the feature description. Each cell contains the frequency with which the keyword of its row appears in the query.

The last step of LSI generates the results. In this step of this approach, we only take into account the model that presents the best similarity measure. We consider it as the most relevant model for the feature to be located, and as such, it is used as input for the next step.

5.6.2 Information Retrieval at model level

Finally, in step 3: Information Retrieval at model level, our approach assesses each model element of the model obtained in step 2. We adapted each step of the LSI technique to work at model level. The adaptation is performed as follows:

Creating a corpus. In this step, each document corresponds to a model element of the model. Each document (model element) includes text from the names of the attributes and operations.

Preprocessing. In this step, we apply the same techniques than in the previous step. The type of the attributes and the type of the parameters in the methods are

removed. Then, all the identifiers are split; for example, “PowerLimit” becomes “power” and “limit”.

Indexing. In this step, in the term-by-document co-occurrence matrix, the terms (rows) correspond to the names of the attributes or operations (i.e., intensity) of the model from the previous step and the documents (columns) correspond to the model elements that are in the model from the previous step.

Model Elements

Terms	ME1	ME2	ME3	ME4	...	MEn	Query
size	6	12	9	6	...	12	6
provider	12	6	4	0	...	24	10
coil	36	30	0	0	...	0	0
small	24	12	2	0	...	6	4
intensity	0	8	10	2	...	0	6
high	6	0	2	0	...	6	8
...	...	...	...	...	...	...	0
name	8	2	1	12	...	3	0

LSI Results

Figure 5.4: Information Retrieval via Latent Semantic Indexing (LSI)

Figure 5.4 shows a term-by-document co-occurrence matrix. Each row in the matrix stands for each one of the unique words (terms) extracted from the runtime model. Figure 5.4 shows a set of representative keywords in the domain such as ’provider’, ’coil’, or ’intensity’ as the terms of each row. Each column in the matrix stands for the model elements of the runtime model. Figure 5.4 also shows the model elements in the columns, which represent the model elements of the runtime model. Each cell in the matrix contains the frequency with which the keyword of its row appears in the document denoted by its column. For instance, in Figure 5.4, the term ’size’ appears 9 times in the ’ME3’ model element.

Querying. In this step, we use the same feature description used in the previous step. Only the relevant terms are taken into account, and words such as determinants and connectors from the language are avoided.

In Figure 5.4, the query column represents the words that appear in the description. Each cell contains the frequency with which the keyword of its row appears in the query. For instance, the term ’provider’ appears 10 times in the query.

A three-dimensional graph of the LSI results is provided in Figure 5.4. The graph shows the representation of each one of the vectors, labeled with letters that

represent the names of the model elements, which are referenced in the box below the graph. The graph reflects the 'ME3' model element vector as being the closest to the query vector, followed by the 'ME1' model element vector.

5.7 Output

After applying all the three steps of our approach, the output produced is a ranking where each model element has been assigned with a value. Only those model elements that have a similarity measure greater than x must be taken into account to measure the quality of the results. A good heuristic that is widely used is $x = 0.7$. This value corresponds to a 45% angle between the corresponding vectors. This threshold has yielded good results in other similar works [35, 79]. Determining a more generally usable heuristic for the selection of the appropriate threshold is an issue under study, over which further research is needed.

The goal of our approach is to rank the relevant model elements within the top positions. The ranking of model elements is ordered by the values of the cosines.

6

BUG LOCALIZATION IN MODELS WITH AN EVOLUTIONARY ALGORITHM

Contents

6.1 Overview **54**

6.2 Approach **54**

6.3 Input **55**

6.4 Population **55**

6.5 Search Strategy **56**

6.5.1 BLiMEA selection 56

6.5.2 BLiMEA crossover 57

6.5.3 BLiMEA mutation 57

6.6 Assessment **59**

6.6.1 Model fragment similarity to the bug description . . 59

6.6.2 The most recent model modification 60

6.7 Output **61**

6.1 Overview

This chapter presents BLiMEA (see Figure 6.1), our bug localization approach in models [11]. This approach uses a Multi-objective Evolutionary Algorithm (MOEA) with two fitness functions: (1) Information Retrieval (IR), and (2) modification timespan. The consideration of timespans is based on the Defect Localization Principle (DLP). This principle is based on the observation that the most recent modifications to a project are most likely the cause of future bugs [80, 81].

The use of a MOEA allows to show the results of both objectives (similarity and modification timespan). The effectiveness of each objective is different for each bug localization. Sometimes the similarity will be more successful to find the model fragment that contains the bug and sometimes the timespan is the most successful.

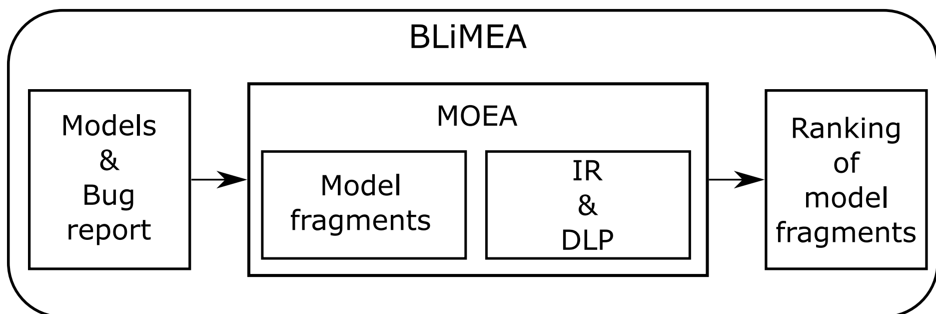


Figure 6.1: Overview of BLiMEA

As a result, software engineers obtain a ranked list of model fragments from the model, intended to identify the parts of the model that are relevant for the bug. Next sections describe the details of the approach.

6.2 Approach

Figure 6.2 presents the details of the BLiM approach for bug location. The left of Figure 6.2 shows the inputs for the approach: a bug description and a set of product models.

The approach relies on an evolutionary algorithm. The center of Figure 6.2 shows a simplified representation of the main steps. The **'Initialize Population'** step calculates an initial population of model fragments from the input set of product models. The **'Genetic Operations'** produce the new generation of model frag-

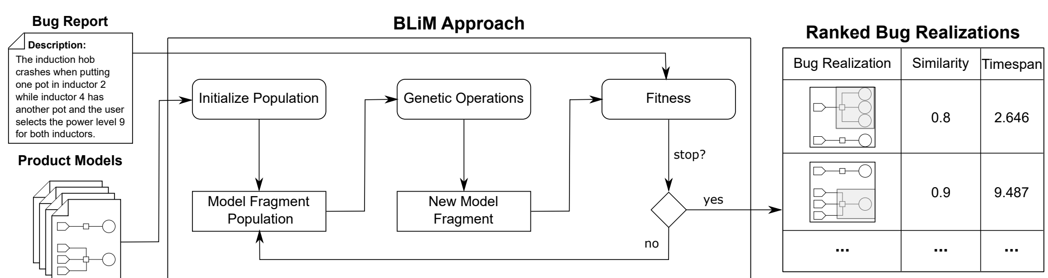


Figure 6.2: The Bug Location Approach in Models: BLiMEA

ments. Finally, the **'Fitness'** step assigns values that assess how good each model fragment is in the following terms: bug description and modification timespan. As output, the approach provides a list of model fragments that might realize the bug (see the right part of Figure 6.2).

6.3 Input

The approach receives as input a **bug description** and a **set of product models**. Typically, the bug description come from textual documentation of a bug report. Therefore, this description will include some domain specific terms that are similar to those used when specifying the product models. The knowledge of the engineers about domain and the product models will be useful for selecting the description from the bug report.

In addition, the software engineer has to select a set of product models from the entire family of products. The software engineer has to use his knowledge to select those models that contain the bug to be located.

6.4 Population

The initial population is generated in the initialize population step of the approach (see Figure 6.2). This step calculates an initial set of model fragments from the input set of product models. This initial set of model fragments is randomly extracted from the product models. This is a common practice in evolutionary computation; as an alternative, seeds (fragments of a model chosen manually) can be proportioned in order to optimize the population.

In evolutionary algorithms, each individual of the population needs to be encoded so the genetic operations can be applied to them. There are different encod-

ings that can be used, although the most common is to encode each individual as a fixed-size string of binary values. The individuals of our population are model fragments, then, the encoding must be able to represent a model fragment extracted from a product model.

We chose a binary encoding to represent our individuals (our model fragments). Each position of the binary string corresponds to a particular element. This element may be or not part of the solution and has two possible values 0 or 1. To encode a model fragment, each position of the binary string represents one model element of the parent model. In other words, each individual will have that position at 0 to indicate that the element is not part of the model fragment or at 1 to indicate that it is part of the model fragment. Thus, we can indicate the subset of element from the parent model that are part of the model fragment.

6.5 Search Strategy

The generation of subsequent populations is performed by applying genetic operators. First, a selection operation selects the model fragments that will be used as parent of the new model fragments. Second, other operations are applied to manipulate the model fragments. In other words, new model fragment sets are generated from existing ones through the use of genetic operators: selection, crossover, and mutation. These operations are described in the following subsections.

6.5.1 BLiMEA selection

Selection is the first step when an evolutionary algorithm is used. This operation selects a number of individuals that will act as parents for the next population. The number of parents depends on the operations that will be performed later, but typically is two. The selection is based on the assessment of the individuals. These individuals will be manipulated later in order to create new individuals with the hope that they will be better solutions than their predecessors.

There is a risk of not exploring areas of the search space that could provide better results. The algorithm may suffer a premature convergence if the individuals with the best fitness are always the only ones selected. The most frequent strategy to cope with this issue is to use a selection mechanism that ensures a proper distribution of the selection [82].

Therefore, the algorithm has to use a selection mechanism where fitter individuals are selected more times than others but that also mitigates the premature

convergence issue. The most usual option is to follow the wheel selection mechanism [83]. That is, each model fragment from the population has a probability of being selected proportional to its fitness score. Therefore, candidates with high fitness values will have higher probabilities of being chosen as parents for the next generation.

6.5.2 BLiMEA crossover

In our encoding, the elements that can be mapped across the different individuals are the model fragment and the referenced product model. This crossover operator will take the model fragment from the first parent and the product model from the second parent. Hence, a new model fragment that contains elements from both parents is generated.

Figure 6.3 shows an example of the crossover operation of BLiMEA. First, the model fragment from the first parent is selected. Second, the product model from the second parent is selected. Then, the operation compares both elements trying to find the model fragment from the first parent in the product model from the second parent. If the comparison finds the model fragment in the product model, the operation creates a new model fragment with the model fragment taken from the first parent but referencing the product model from the second parent. In the case that the comparison does not find the model fragment from the first parent in the second parent, the operation will return the first parent unchanged.

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

6.5.3 BLiMEA mutation

As we said, each model fragment is referencing a product model. This operation mutates each model fragment in the context of its referenced product model. In other words, the model fragment will gain or drop some elements, but the resulting model fragment will still be part of the referenced product model. The mutation possibilities of a given model fragment are driven by its associated product model. Figure 6.3 shows an example of the mutation operation of BLiMEA.

The type of mutation, addition or removal of elements, is decided randomly:

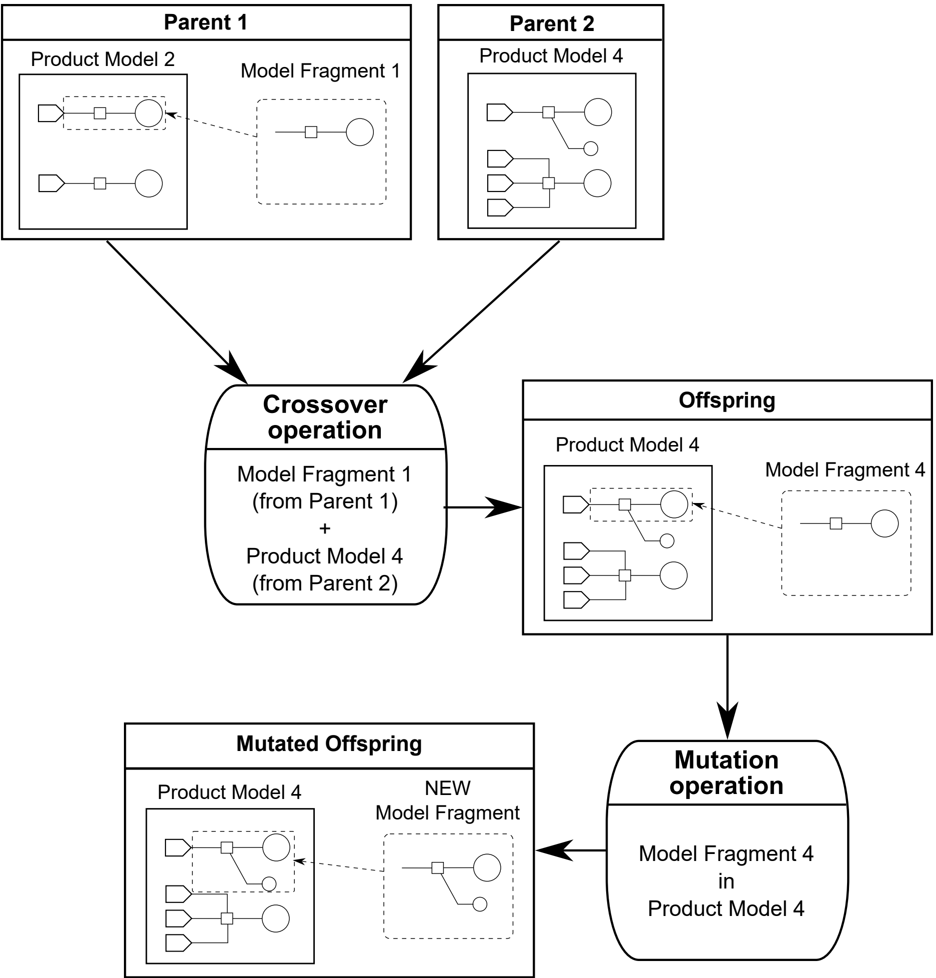


Figure 6.3: Crossover and mutation operations of BLiMEA

- The subtractive mutation randomly removes some elements from the model fragment. The resulting model fragment have to be connected, cannot be split into several isolated groups of elements. Hence, the only constraint is that the elements to be removed are the ones that are on the edges of the model fragment. Since the resulting model fragment is a subset of the original model fragment and the original was present in the referenced product model, the resulting product model will always be present in the referenced product model.

- The additive mutation randomly adds some elements to the model fragment. In the same way as in the previous type, the resulting model fragment has to

be part of the referenced product model. To do that, the operation identifies the boundaries of the model fragment and then a random element from the boundary is added to the model fragment. The resulting model fragment will be part of the referenced product model.

The result of the application of these operations is a new model fragment. This new model fragment represents another possible solution that can contain the bug for the specific bug being located.

6.6 Assessment

In evolutionary algorithms, the fitness step is used to assess the different degrees of adaptation to the environment that different model fragments have. Following this idea, our fitness step is used to determine the suitability of each model fragment to the problem. The input of this step is a set of model fragments, and the output produced is a set where each model fragment has been assigned with two fitness values: the similarity to the bug description, and the timespan to the most recent model fragment modifications.

6.6.1 Model fragment similarity to the bug description

Similar to previous chapters, we apply a information retrieval technique, Latent Semantic Indexing (LSI), to analyze the relationship between the description of the bug description and the model fragment.

To build the term-by-document co-occurrence matrix, the model fragments and the bug description are used. The documents are text representations of the model fragments. The text of each document corresponds to the names and values of the attributes and operations of each model fragment. The query is constructed from the terms that appear in the bug description. If the terms used for the model and for the bug description differ too much, LSI will not work. Therefore, the text from the documents (model elements) and the text from the query (bug description) are homogenized by applying well-known Natural Language Processing techniques (tokenizing, Parts-of-Speech Tagging, and Lemmatizing) to reduce this gap. If the languages used differ too much, other techniques such as manual annotation of the model elements could be applied at the expense of increasing the effort. The union of all the words extracted from the documents (model fragments) and from the query (bug description) are the terms (rows) used by our LSI fitness.

Once the matrix is built, we normalize and decompose it into a set of vectors using a matrix factorization technique called Singular Value Decomposition (SVD) [77]. One vector that represents the latent semantics of the document is obtained for each model fragment and the query. Finally, the similarities between the query and each model fragment are calculated as the cosine between the two vectors. The fitness value that is given to each model fragment is the one that we obtain when we calculate the similarity, obtaining values between -1 and 1.

6.6.2 The most recent model modification

To apply the Defect Localization Principle, we measure the timespan to the last modification of the model. As this has not been applied before for bug localization in models, we proposed 4 different ways to apply it in models, these forms are explained in [11]. However, in this summary we only show the assessment that achieved the best results: the most recent model modification timespan.

As the modifications affect the model elements, each model element has its own modification timespan. Thus, each model element has a constant value of modification timespan during the execution of the algorithm, but different model elements have different modification timespan values.

The modifications of the model over time are considered when extracting the most relevant model fragment for the target bug (the time difference between the last modification of a model element and the usage day). A recently modified model element (i.e., a short timespan) has a lower timespan value than another model element that was modified farther in the past. Since a model fragment is composed by a set of model elements, the timespan weighting of the model fragment depends on the timespan weightings of the model elements that compose it.

The time difference is based on the number of days and can therefore be very large when the model fragment was modified a long time ago. To normalize the time difference, mathematical solutions such as square root or logarithm can be used. We decided to use square root because it has achieved good results in other works that use time differences [84, 85]. To use of the square root is more suitable and more effective for the proposed approach [81].

Figure 6.4 shows an example of the application of this function to a model fragment. This function expresses the concern of capturing primarily the model fragments with the model elements that have the lowest modification timespans. That is, model elements that have been recently modified. Then, the value of the

Figure 6.4: Timespan of the modifications of the model elements of a fragment

model fragment will be the value of the most recently modified model element. In the example of Figure 6.4, the timespan value of the model fragment is 7 days, that means a square root of 2.646.

6.7 Output

The output of BLiMEA (see Figure 6.2) is an ordered set of model fragments that contains the target bug. The software engineer obtains this set of model fragments, which is intended to identify the parts of the model that are relevant to the bug.

To do so, the engineer can order the ranking following different criteria: the similarity to the bug description, or the most recent model fragment modifications.

7

Evolutionary Algorithm for Bug Localization in the Reconfigurations of Models at Runtime

Contents

7.1 Overview **64**
7.2 Approach **64**
7.3 Input **65**
7.4 Population **65**
7.5 Search Strategy **66**
7.5.1 EBRo selection 66
7.5.2 EBRo crossover 66
7.5.3 EBRo mutation 68
7.6 Assessment **68**
7.7 Output **70**

7.1 Overview

This chapter presents EBRo (see Figure 7.1), our approach for bug localization in reconfigurations [13]. In systems with models at runtime, the models experience reconfigurations at runtime due to context changes, being these reconfigurations a source of bugs. The development of this approach was focused on locating bugs that appear as the result of dynamic reconfigurations of the systems due to context changes.

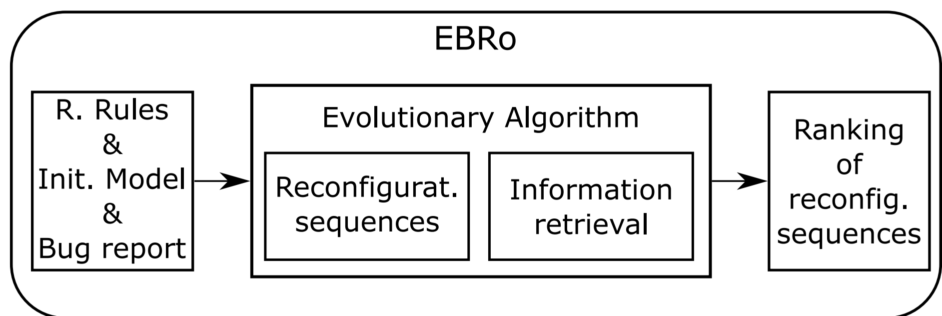


Figure 7.1: Overview of EBRO

This approach uses an evolutionary algorithm guided by a fitness function that considers the similarity to the description of the bug. To measure the textual similarity, we start from an initial model at runtime to which we apply a sequence of reconfigurations, obtaining another model in which we evaluate whether the modified elements are similar with the description of the bug. As a result, software engineers obtain a ranked list of reconfiguration sequences, intended to identify the reconfiguration rules that are relevant to the bug. Next sections describe the details of the approach.

7.2 Approach

The general structure of our approach (EBRo) is introduced in Figure 7.2. The goal of EBRo is to obtain a ranked list of sequences of reconfigurations rules from a given list of reconfiguration rules that may trigger the bug specified by the bug description. Our EBRo approach takes as input a set of reconfiguration rules, an initial model, and a bug description.

The output of EBRo (see Figure 7.2) is a set of reconfiguration sequences. The search space for our approach is determined not only by the number of triggered

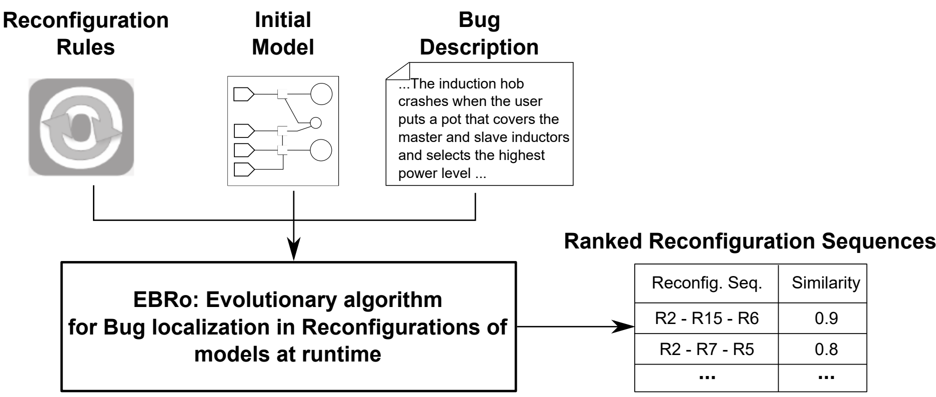


Figure 7.2: Input and output of our bug localization in reconfigurations of models at runtime

reconfigurations, but also by the order in which they are applied. To explore the search space, EBRo uses an evolutionary algorithm that enables the exploration of a large number of possible reconfigurations.

7.3 Input

The approach receives as input a set of reconfiguration rules, an initial model, and a bug description. The set of reconfiguration rules describe the changes in the model at runtime. The reconfiguration rules are triggered by context changes. The initial model is the model that specifies the initial configuration. Finally, the bug description of the bug that we want to locate using natural language. This bug description is obtained in the same way we explain in the previous chapter (see Section 6.3).

7.4 Population

Each individual of the population is a reconfiguration sequence. To represent each individual, we used a vector representation. Each vector's dimension represents a reconfiguration rule. Thus, a solution is defined as a sequence of reconfigurations applied to a model. The size of the solution represents the number of reconfigurations (dimensions) in the vector. When created, the order of the reconfigurations corresponds to their positions in the vector.

An example of an individual is given in Figure 7.3. This individual contains three dimensions that correspond to three reconfigurations applied to the initial

R1: newChannelForDoubleInductor(upperPowerGroup, secondaryInductor)
R21: redirectPowerFromInverter(secondInverter,upperPowerGroup)
R3: activatePowerFromInverter(secondInverter)

Figure 7.3: Representation of an individual

model. For instance, the predicate *newChannelForDoubleInductor(upperPowerGroup, secondaryInductor)* means that a new *channel* is created in the *upper power group*, connecting it with the *secondary inductor*.

7.5 Search Strategy

To allow the generation of new populations, we have to define a selection of the individuals that will work as parents of the new populations. In addition, we have to define the operators that allow the creation of the new individuals of the population: crossover and mutation.

7.5.1 EBRo selection

To select individuals, we use stochastic universal sampling (SUS) [86]. This technique of selection of an individual is directly proportional to its relative fitness in the population. SUS is a random selection algorithm which gives a higher probability of selection to the fittest solutions while still giving a chance to every solution.

In each iteration of the algorithm, SUS is used to select individuals from the population (P_n) for the next generation of the population (P_{n+1}). The selected individuals will be the ones that generate the next individuals using genetic operations.

7.5.2 EBRo crossover

We use a single, random, cut-point crossover. It starts by selecting and splitting at random two parent solutions. When two parent individuals are selected, a random cut point is determined to split them into two sub-vectors. Then, the crossover

creates two child solutions by putting, for the first child, the first part of the first parent with the second part of the second parent, and, for the second child, the first part of the second parent with the second part of the first parent.

Each solution has a length limit in terms of number of reconfigurations. When applying the crossover operator, the new solution may have the minimum length between the two parents. Then, the crossover operator must enforce the length limit constraint by eliminating some reconfiguration rules.

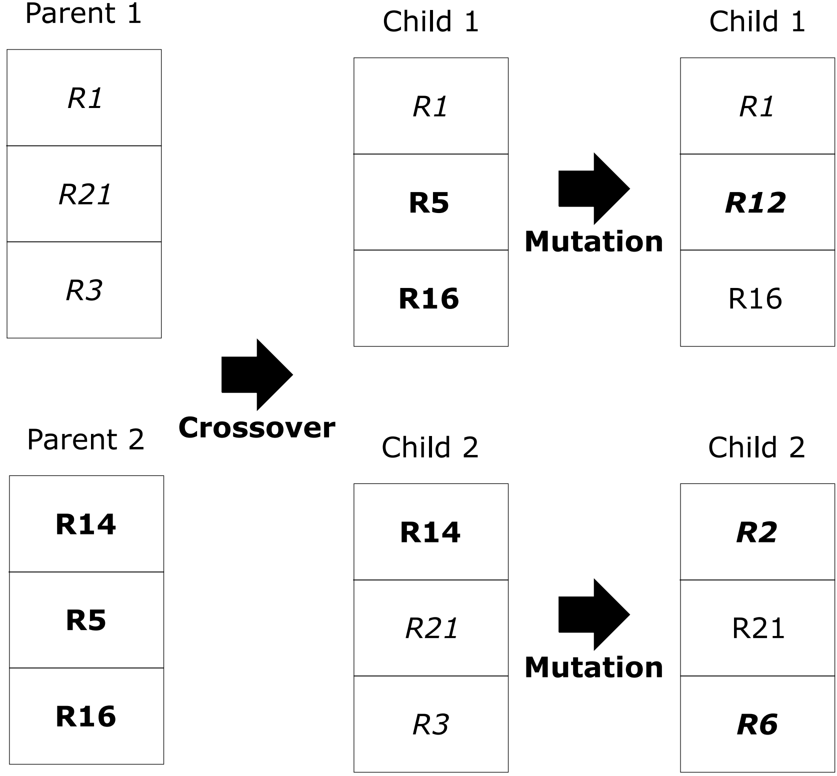


Figure 7.4: Crossover and mutation operators applied to reconfigurations

Figure 7.4 shows an example of applying the crossover operator. In this example, Parent 1 (P_1) and Parent 2 (P_2) are combined to generate two new solutions. The upper sub-vector of P_1 is combined with the bottom sub-vector of P_2 to form Child 1, and the bottom sub-vector of P_1 is combined with the upper sub-vector of P_2 to form Child 2.

7.5.3 EBRo mutation

This operator consists of randomly changing one or more reconfigurations in the vector of reconfigurations. Given an individual, the mutation operator first randomly selects some positions in the vector representation of the individual. Then, the selected dimensions are replaced by other reconfiguration rules.

Figure 7.4 shows an example of applying the mutation operator. In Child 1, the mutation operator replaces dimension number two (*R5* by *R12*), while in Child 2, the mutation operator replaces dimensions number one and three (*R14* and *R3* by *R2* and *R6*).

When creating the sequence of reconfigurations, we do not guarantee that they are feasible and that they can be applied. However, this could be solved by applying some repair operations that are part of our future work.

As a result, new reconfiguration sequences are created. In other words, the new reconfiguration sequences represent other possible solutions that can trigger the bug for the specific bug being located.

Overall, the aim of the approach is to find the most relevant reconfiguration sequence that triggers the bug described by the bug report. To do so, the algorithm of EBRo performs a search guided by a fitness function. This search is done among the different reconfiguration sequences (previously obtained by applying the mutation and crossover operations) that could conform to the bug description.

7.6 Assessment

After creating a solution, it should be assessed using a fitness function. The fitness function quantifies the quality of the proposed reconfiguration sequence. In this approach, we use the information retrieval technique presented in previous chapters, called Latent Semantic Indexing (LSI). EBRo assesses the relevance of each reconfiguration sequence in relation to the bug description provided by the user. The input of this step is a set of reconfiguration sequences, and the output is the set of reconfiguration sequences, where each reconfiguration sequence has been assigned with a fitness value regarding its similarity to the bug description.

Figure 7.5 shows how we extract the texts needed to use LSI. First, we apply the reconfiguration sequence to the initial model configuration. After applying it, we obtain a new model from which we extract the model elements that have been modified by the reconfigurations. In Figure 7.5, the modified model elements are

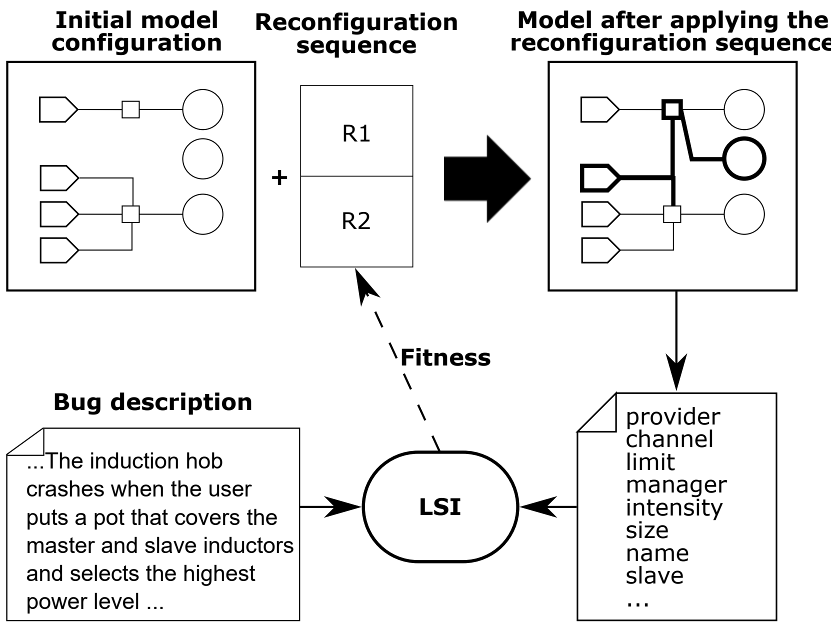


Figure 7.5: Terms extraction from a reconfiguration sequence

the ones in bold. The texts for the LSI documents are the names and values of the properties and methods of each model element.

In this approach, the LSI documents are model elements, i.e., a document of text is generated from the text of the model elements that have been modified by the reconfiguration. The query is constructed from the text that appears in the bug description. To reduce the gap between the text from the model elements and the bug description, this approach uses the same strategy as BLiMEA, Natural Language Processing techniques (tokenizing, Parts-of-Speech Tagging, and Lemmatizing) are applying.

The union of all the keywords extracted from the documents (model elements) and from the query (bug description) are the terms (rows) used by our LSI fitness. Each column is one of the model elements that have been modified by the reconfiguration. The last column is the query obtained from the bug description of the user. Each row is one of the terms extracted from the corpuses of text composed by all of the model elements and the query itself. Each cell has the number of occurrences of each of the terms in the model elements.

Once the matrix is built, we normalize and decompose it into a set of vectors using the same matrix factorization technique as in the previous chapter, see Section 6.6.1.

7.7 Output

The output of EBRo (see Figure 7.2) is an ordered set of reconfiguration sequences that might trigger the target bug. The ranking is ordered following the similarity to the bug description.

Part III

Case Studies and Evaluation

8

Case Studies

Contents

8.1 Overview 74
8.2 Induction Hob Domain 74
8.3 Train Control and Management Domain 74
8.4 Smart Hotel Domain 75

8.1 Overview

This chapter presents the three case studies used to evaluate the approaches presented in this dissertation. First, we present the case studies from our industrial partners. BSH presented previously (see Chapter 4), and CAF a worldwide provider of railway solutions. second, we present the Smart Hotel that is an autonomic system with models at runtime.

8.2 Induction Hob Domain

This case study corresponds to one of our industrial partners, BSH. It is already presented in section 4.3 as the running example. The induction division of BSH has been producing induction hobs under the brands of Bosh and Siemens for the last 15 years.

The case study of BSH is composed of 46 induction hob models where, on average, each product model is composed of more than 500 elements. The documentation includes 96 different features and 37 bug reports. Those features and bugs are in products that are currently being sold or will be released to the market in the near future.

8.3 Train Control and Management Domain

This case study corresponds to another of our industrial partners, CAF which is a worldwide provider of railway solutions. Their trains can be seen all over the world and in different forms (regular trains, subway, light rail, monorail, etc.).

A train unit is furnished with multiple pieces of equipment through its vehicles and cabins. These pieces of equipment are often designed and manufactured by different providers, and their aim is to carry out specific tasks for the train. Some examples of these devices are: the traction equipment, the compressors that feed the brakes, the pantograph that receives power from the overhead wires, or the circuit breaker that isolates or connects the electrical circuits of the train. The control software of the train unit is in charge of making all the equipment cooperate in providing the train with functionality while guaranteeing compliance with the specific regulations of each country.

The DSL of our industrial partner has the required expressiveness to describe the interaction between the main pieces of equipment installed in a train unit.

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

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

In this kind of products, it is important to ensure that no bug is going to occur while the train is running. This task is very cumbersome and time consuming for the domain engineers due to the high number of tests involved.

The product family of CAF is composed of 23 trains where each product model is composed of more than 1200 elements on average. They provide us with documentation of 56 bug reports, the approved reconfiguration sequences that triggers the bug and the model fragments that contain the bugs.

8.4 Smart Hotel Domain

The Smart Hotel [87] is an autonomic system with models at runtime. It is reconfigured in response to changes in the context, for example if there is a client in the room or not, and what activities they may be performing (sleeping, watching TV, ...). This section shows the language used for specifying the architecture model of the Smart Hotel. This section also shows how the architecture model is reconfigured at runtime in response to context changes.

The Smart Hotel is described using Pervasive Modeling Language (PervML) [88]. PervML is a Domain Specific Language (DSL) that describes pervasive systems using high-level abstraction concepts such as services. This language is focused on specifying heterogeneous services in distinct physical environments. This DSL has been applied to develop solutions in the Smart Hotel domain.

The Smart Hotel reconfiguration engine determines how the system should be reconfigured in response to a context change, and then it modifies the architecture model accordingly. In models at runtime, a causal connection between the system and the runtime model is defined (there is a bidirectional relation between the source code and the runtime model). This connection allows the models (usually the architecture model) to reflect the software state. This connection can be

achieved in different ways; however, the most used implementation is the MAPE-K loop [27, 89].

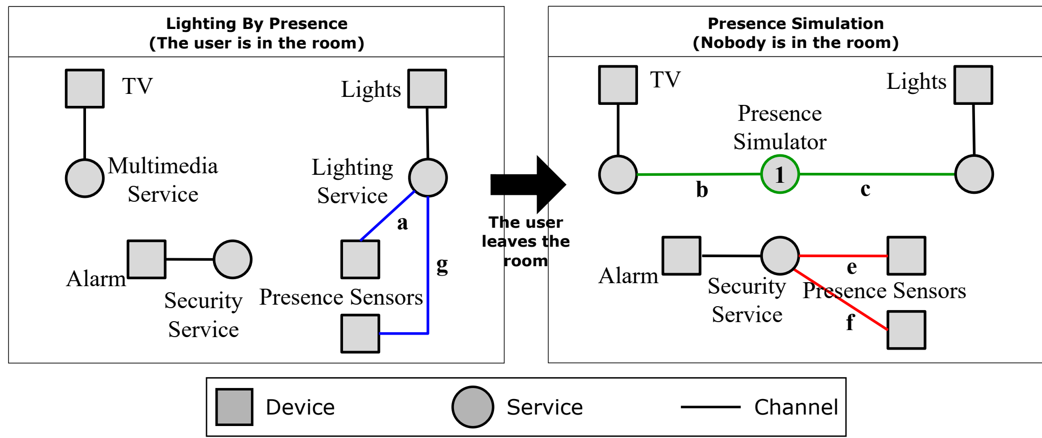


Figure 8.1: Smart Hotel Model Reconfigurations

Figure 8.1 shows two Smart Hotel configurations according to the concrete syntax of the architecture model of PervML [88]. Figure 8.1 (left) shows a *User in the room* configuration, while Figure 8.1 (right) shows a *Nobody in the room* configuration. It can be observed that movement sensors are used for different purposes: lighting (left) and providing information to the security service (right). In addition, the Occupancy simulation service is activated in the *Nobody in the room* configuration, and the connections that are required for this service to communicate with multimedia, lighting, and security services are established.

9

EVALUATIONS

Contents

9.1 Overview **78**

9.2 Oracle **78**

9.3 Comparison and Measure **79**

9.4 Measurements and Statistical Analysis **80**

9.4.1 Measurements derived from the comparison 80

9.4.2 Statistical analysis of the measurement results 81

9.5 Results **83**

9.5.1 DFL Evaluation 83

9.5.2 BLiMEA Evaluation 85

9.5.3 EBRo Evaluation 86

9.1 Overview

This chapter presents the evaluation process followed to test our approaches. We describe each of the elements involved in the evaluation process: the oracle preparation, the measurements and statistical analysis performed, and the results obtained.

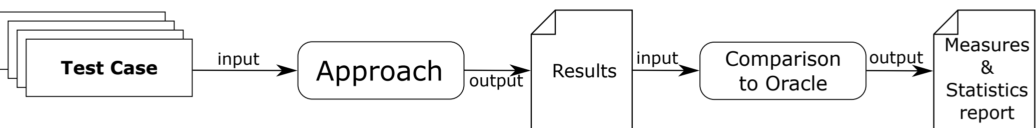


Figure 9.1: Evaluation process for each of the approaches

Figure 9.1 shows an overview of the process followed to evaluate each of the approaches. The test cases are the input for the approach under evaluation. After running the approach, we obtain the results. These results are compared with an oracle in order to check the accuracy. Finally, we report the quality and the statistical analysis of the results.

9.2 Oracle

The oracle is the mechanism that we used to evaluate the results of our approaches. The oracle is considered the ground truth and is used to compare the results of the approaches with the oracle. Hence, the oracle is composed by a set of product models and a set of features and bugs already located. Thus, we know beforehand the solutions for each of the features or bugs that we want to locate. The information that compose the oracle is provided by our industrial partners and contains all the domain knowledge needed: features realization, model elements that contain bugs, feature descriptions, and bug reports.

Although we have illustrated the approaches with the running example of BSH. We have evaluated our approach with the three different case studies. Table 9.1 summarizes the domain in which we have evaluated each of the approach. DFL was evaluated in the Smart Hotel, BLiMEA was evaluated in BSH, and EBRo was evaluated in BSH and CAF.

The Smart Hotel is composed by 476 model elements in the architecture model. It is configured with a feature model. The feature model specifies the 39 different features that the Smart Hotel has implemented.

	DFL	BLiMEA	EBRo
Smart Hotel	X		
BSH		X	X
CAF			X

Table 9.1: Approach and evaluation domains

The product family of BSH is composed of 46 induction hob models, which, on average, are composed of more than 500 elements. Our industrial partner provided us with documentation of 37 bug reports, the reconfigurations (when needed) and the approved model fragments that contain the bugs. The approved model fragments have between 3 and 15 model elements, with an average of 8 model elements. It is important to highlight that each model element has properties (that include terms), and all the information needed, such as the modification timespan. Five domain engineers from our industrial partner were involved in providing the set of 37 bugs. The domain engineers of BSH based their selection on a combination of importance and frequency. The set of bugs provided are the most representatives of the bugs that occurs in BSH.

The product family of CAF is composed of 23 trains where each product model is composed of more than 1200 elements on average. They provide us with documentation of 56 bug reports, the approved reconfiguration sequences that triggers the bug and the model fragments that contain the bugs. In the same way as in BSH, the domain engineers were involved throughout the process and the set of bugs are representatives of the ones that occur in CAF.

9.3 Comparison and Measure

Once we have the results of the approaches, we have to compare it with the oracle and measure them in terms of some software quality properties. To compare them, we used a confusion matrix [90].

A confusion matrix is a table that is often used to describe the performance of a classification model (in this case, the approach under evaluation) on a set of test data (the solutions) for which the true values are known (from the oracles). In our case, each solution that is output by the approaches is a subset of the elements that are present in the product model (where the feature of the bug is being located). Since the granularity will be at the level of these elements, the presence or absence

of each element will be considered as a classification. Therefore, our confusion matrices will distinguish between two values (TRUE or presence and FALSE or absence).

We obtain a confusion matrix for each of the solutions predicted by each of the approaches. The confusion matrix arranges the results of the comparison into four categories:

True positive (TP): An element presents in the predicted solution that is also present in the actual solution.

True negative (TN): An element not present in the predicted solution that is not present in the actual solution.

False positive (FP): An element presents in the predicted solution that is not present in the actual solution.

False negative (FN): An element not present in the predicted solution that is present in the actual solution.

The confusion matrix holds the results of the comparison between the predicted solutions and the actual solutions. The result of the sum of all the categories (TP+TN+FP+FN) is the number of elements that contains the predicted solution. However, in order to evaluate the performance of the approach, it is necessary to extract some measurements from the confusion matrix.

9.4 Measurements and Statistical Analysis

In this section, we present the quality measurements derived from the confusion matrix and the statistical analysis performed. This analysis provides quantitative evidence of the impact of our approaches and shows that this impact is significant.

9.4.1 Measurements derived from the comparison

Some performance measurements are derived from the values in the confusion matrix. We use four measurements, these measurements are: recall, precision, F-measure and Matthews Correlation Coefficient (MCC).

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

$$Recall = \frac{TP}{TP + FN} \tag{9.1}$$

Precision measures the number of elements from the predicted solution that are correct according to the actual solution and is defined as follows:

$$Precision = \frac{TP}{TP + FP} \tag{9.2}$$

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

$$F - measure = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{9.3}$$

Recall values can range between 0% (i.e., no single element from the actual solution is present in the predicted solution) to 100% (i.e., all the elements from the actual solution are present in the predicted solution).

Precision values can range between 0% (i.e., no single element from the predicted solution is present in the actual solution) to 100% (i.e., all the element from the predicted solution are present in the actual solution). A value of 100% precision and 100% recall implies that both the predicted solution and the actual solution are the same.

However, none of these 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). MCC is a balanced measure which can be used even if the search space and the predicted solution are of very different sizes [91]. For this reason, MCC is one of the best measures for describing a confusion matrix [92]. It is defined as follows:

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

9.4.2 Statistical analysis of the measurement results

To properly compare our approaches, all of the data resulting from the empirical analysis was analyzed using statistical methods following the guidelines in [93]. The goals of our statistical analysis are: (1) to provide formal and quantitative

evidence (statistical significance) that the approach under evaluation does in fact have an impact on the comparison metrics (i.e., that the differences in the results were not obtained by mere chance); and (2) to show that those differences are significant in practice (effect size).

Statistical Significance

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

The statistical tests provide a probability value, $p - Value$. The $p - Value$ 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 [93], so the hypothesis H_0 can be considered false.

The test that we must follow depends on the properties of the data. Since our data does not follow a normal distribution in general, our analysis requires the use of non-parametric techniques. There are several tests for analyzing this kind of data; however, the Quade test shows that it is more powerful than the others when working with real data [94]. In addition, according to Conover [95], the Quade test has shown better results than the others when the number of algorithms is low (no more than 4 or 5 algorithms).

However, with the Quade test, we cannot answer the following question: Which of the algorithms gives the best performance? In this case, the performance of each algorithm should be individually compared against all other alternatives. In order to do this, we perform an additional post hoc analysis. This kind of analysis performs a pair-wise comparison among the results of each algorithm, determining whether statistically significant differences exist among the results of a specific pair of algorithms.

Effect size

When comparing algorithms with a large enough number of runs, statistically significant differences can be obtained even if they are so small as to be of no practical value [96]. Thus, it is important to assess if an algorithm is statistically better than another and to assess the magnitude of the improvement. Effect size measures are needed to analyze this.

For a non-parametric effect size measure, we use Vargha and Delaney's $\hat{A}_{12}$ [97, 98]. $\hat{A}_{12}$ measures the probability that running one algorithm yields higher values than running another algorithm. If the two algorithms are equivalent, then $\hat{A}_{12}$ will be 0.5.

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

9.5 Results

This section presents the results obtained in the evaluation of each of the approaches presented in this dissertation.

9.5.1 DFL Evaluation

This evaluation has to goals: (1) evaluate whether our dynamic feature localization approach applied to models outperforms a current approach applied to program code, and (2) evaluate whether all the changes produced in the model at runtime are relevant for feature localization.

The first evaluation is part of the work of SAM '16 [9]. The experiment evaluates whether our dynamic feature localization approach with models at runtime achieves better results than a current approach [34] that uses program code to perform feature localization (SCFL). We choose the approach presented in [34] because is the one that shows the best results for feature location in source code [3, 4].

Our DFL approach and the SCFL approach used the Smart Hotel runtime model and source code traces, respectively. Our DFL approach produced a ranking of model elements and the SCFL approach produced a ranking of methods for

the targeted feature. The feature model oracle enables us to know how many of the model elements or methods in the ranking were the ones that belongs to the target feature.

Our DFL approach ranks the relevant elements in the top ten positions of the ranking in 84% of the cases. In the top positions of the ranking of the SCFL approach, there are false positives associated with some programming patterns and true positives are spread between position 15 and 100.

It is accepted by the feature localization community [34, 36] that, a feature localization approach is considered better than another feature localization approach when it produces a ranking where the elements that belong to the feature are in higher positions than in the ranking of the other approach. In our evaluation, our DFL approach obtained better positions in the ranking than the SCFL approach.

The second evaluation is part of the work of ECMFA '17 [10]. We evaluate whether all the changes produced in the model when a system reconfiguration is necessary are relevant for feature location. In order to do this, we compare the presented Dynamic Feature Location approach using traces following the architecture criterion (DFL-AT), against the same approach using traces following the configuration criterion (DFL-CT).

In the configuration criterion, the snapshots are added to the trace when the runtime model corresponds to a target configuration of the system in a reconfiguration. In the architecture criterion, the snapshots are added to the trace when a change in the runtime model is performed. That is, the traces recorded following the architecture criterion contain more information than the ones recorded following the configuration criterion.

	DFL-AT	DFL-CT
Recall	0.74	0.64
Precision	0.75	0.65
F-measure	0.74	0.63

Table 9.2: Mean values for precision, recall, and F-Measure for each execution in the Smart Hotel

Table 9.2 shows the recall, precision, and F-measure values. On average, DFL-AT obtains a value of 0.74 in recall while DFL-CT obtains a value of 0.64. DFL-AT improves the recall result achieved by DFL-CT by around 10%. Regarding the precision value, on average, DFL-AT obtains a 0.75 while DFL-CT obtains a 0.65. Once again, DFL-AT improves the precision result achieved by DFL-CT

by around 10%. Consequently, on average, DFL-AT obtains a value of 0.74 in F-measure, while DFL-CT obtains a 0.63. Taking into account the results, DFL-AT outperforms the results of DFL-CT.

9.5.2 BLiMEA Evaluation

This evaluation is part of the work of ISD '17 [11]. The goal of that evaluation was the evaluation of different timespan weighting objective functions. However, in this dissertation, we only show the results obtained by the best function.

We compare the BLIMEA approach with a baseline [99]. The baseline is the approach used by our industrial partner for bug localization. Even though it was designed with other purpose in mind (feature localization), said approach is the best bug localization technique available to our industrial partner. To perform the evaluation, we applied the approaches to one of our case studies, BSH.

	BLiMEA	Baseline
Recall $\pm$ (σ)	0.79 $\pm$ 0.11	0.44 $\pm$ 0.14
Precision $\pm$ (σ)	0.73 $\pm$ 0.09	0.29 $\pm$ 0.09
F-measure $\pm$ (σ)	0.76 $\pm$ 0.08	0.35 $\pm$ 0.09

Table 9.3: Mean values and standard deviations for precision, recall, F-Measure and MCC for each approach in BSH

Table 9.3 shows the mean values of recall, precision, and F-measure for both BLiMEA and the baseline for the case study. BLiMEA obtains the best results, providing an average value of 0.79 in recall and 0.73 in precision. The baseline obtains an average value of 0.44 in recall and 0.29 in precision.

In the Quade test statistical analysis, the $p-values$ obtained were $\ll 2x10^{-16}$ for recall and precision. Since the $p-values$ are smaller than 0.05, we reject the null hypothesis. Consequently, we can state that there exist differences between the two approaches for the performance indicators of recall and precision. In addition, the same value for the $p-value$ were obtained in the Holm's post hoc analysis. This indicates that the differences of performance between the two approaches are significant.

In the effect size statistics, BLiMEA achieves better recall than the baseline 96% of the times and better precision almost all the times. BLiMEA obtained the best performance results between the two evaluated approaches (see Table 9.3).

Overall, these results confirm that the use of BLiMEA against the baseline approach has an actual impact. However, we can see that there are some bugs (around 24% on average) that are not properly located by the approach. This happens because the fitness functions that guides the search is not giving high fitness values to the model fragments realizing those bugs. This can happen due to differences between the language used in the bug descriptions and the product models, or in cases where there are few differences in the modification timespan among the different model fragments.

9.5.3 EBRo Evaluation

This evaluation is part of the work of MoDELS '18 [13]. The experiment evaluates whether or not the information found in the reconfiguration sequences improves the bug localization results.

We compare the EBRo approach with the baseline [99] and with a random search (RS). The baseline is the same as in the previous section, the approach used by our industrial partners for bug localization. In addition, RS works as sanity check. If RS outperforms an intelligent search method, we can conclude that there is no need to use a metaheuristic search. To perform the evaluation, we applied the approaches to two of our case studies: BSH and CAF.

		EBRo	Baseline	RS
BSH	Recall $\pm$ (σ)	0.83 ± 0.14	0.72 ± 0.14	0.35 ± 0.04
	Precision $\pm$ (σ)	0.78 ± 0.09	0.63 ± 0.09	0.40 ± 0.06
	F-measure $\pm$ (σ)	0.79 ± 0.09	0.66 ± 0.07	0.37 ± 0.03
	MCC $\pm$ (σ)	0.78 ± 0.10	0.63 ± 0.08	0.31 ± 0.04
CAF	Recall $\pm$ (σ)	0.80 ± 0.08	0.70 ± 0.09	0.41 ± 0.05
	Precision $\pm$ (σ)	0.76 ± 0.12	0.65 ± 0.11	0.38 ± 0.06
	F-measure $\pm$ (σ)	0.77 ± 0.07	0.67 ± 0.07	0.39 ± 0.04
	MCC $\pm$ (σ)	0.75 ± 0.08	0.64 ± 0.08	0.33 ± 0.05

Table 9.4: Mean values and standard deviations for precision, recall, F-Measure and MCC for each approach and each case study

Table 9.4 shows the mean values of recall, precision, F-measure, and MCC for the EBRo, baseline, and RS approaches in each case study. The EBRo approach obtains the best results in recall, precision, and MCC, providing an average value

of 0.83 in BSH and 0.80 in CAF in recall, 0.78 in BSH and 0.76 in CAF in precision, and 0.78 in BSH and 0.75 in CAF in MCC. The second-best results are obtained by the baseline, providing an average value of 0.72 in BSH and 0.70 in CAF in recall, 0.63 in BSH and 0.65 in CAF in precision, and 0.63 in BSH and 0.64 in CAF in MCC. The RS approach provided an average value of 0.35 in BSH and 0.41 in CAF in recall, 0.40 in BSH and 0.38 in CAF in precision, and 0.31 in BSH and 0.33 in CAF in MCC. In terms of recall, precision, and MCC, EBRo outperforms the rest of the approaches.

Regarding the statistical analysis, the $p-Values$ obtained in the Quade test are well below 0.05. Consequently, we can state that there are significant differences among the algorithms for the four performance indicators. In addition, all of the $p - Values$ of Holm's post hoc analysis are smaller than their corresponding significance threshold value (0.05), indicating that the differences in performance between the algorithms are significant.

In the effect size statistics, the largest differences were obtained between EBRo and the RS approach, where EBRo achieves better results all of the times. When comparing EBRo and the baseline, the differences are not so large, with EBRo achieving better results in around 80% of the times. EBRo obtained the best performance results among the three evaluated approaches (see Table 9.4.

The performed statistical analysis indicated that EBRo outperforms the rest of the approaches in terms of recall, precision, and MCC. Overall, these results confirm that the use of EBRo against the baseline and the RS approaches has an actual impact.

Our results confirm that the EBRo and the baseline approaches are better than random search based on the four metrics (recall, precision, F-measure, and MCC) on both the BSH and CAF case studies. Through this study, we concluded that there is empirical evidence to support the significance of the results of our algorithm. Thus, an intelligent algorithm is required to find good solutions to perform bug localization in reconfigurations of models at runtime.

In addition, the solutions obtained with the EBRo approach are better than the ones obtained with the baseline approach. The search space with the EBRo approach is driven by the reconfigurations, as it happens at runtime. While the baseline explores a much larger search space, any model conforms to the metamodel, and does not take into account the runtime reconfigurations.

Part IV

Discussion and Conclusion

10

DISCUSSION

Contents

10.1 Overview 92

10.2 Issues detected in our approaches 94

10.2.1 Vocabulary mismatch 94

10.2.2 Implicit knowledge 94

10.2.3 Invalid reconfiguration sequences 95

10.2.4 Poor use of models 95

10.2.5 Bugs not related to timespan modifications 96

10.2.6 Poor model traces 96

10.3 Findings detected in our approaches 97

10.3.1 Reduction of search space 97

10.3.2 High level of abstraction 97

10.3.3 Good performance between information retrieval and the defect localization principle 98

10.3.4 Non-reliance on the domain 98

10.4 Comparison to other works 98

10.1 Overview

Based on the experiences from the evaluations performed, we present the lessons that we learned in order to assist researchers in the context of feature and bug localization in models.

We have applied our approaches to several model-based systems. The left part of Figure 10.1 shows the systems and how they are related to each other. This thesis focuses on model-based systems. The right part of Figure 10.1 shows the elements that appear in each of the systems. For example, all model-based systems have design time models. However, not all model-based systems have models at runtime or dynamic reconfigurations.

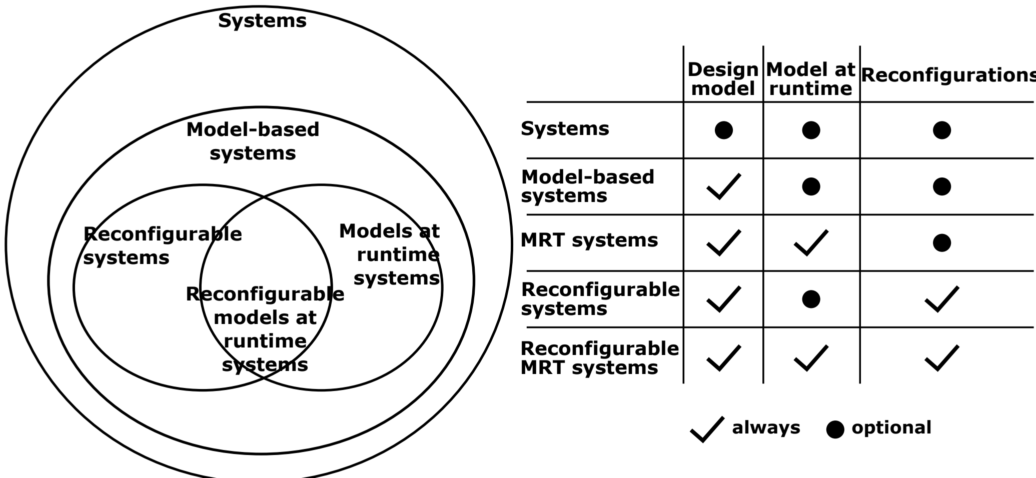


Figure 10.1: Systems in which we have applied our approaches

The approaches presented throughout this thesis need some model elements in order to work (see Table 10.1). To perform bug localization with EBRo, we need a system with at least design time models and dynamic reconfigurations. To perform bug localization with BLiMEA, we need a system with at least design time models. And to perform feature localization with DFL, we need a system with at least, design time models and runtime models.

In addition, taking into account the results obtained in our evaluations, we realized that our approaches are complementary. In the real word, before locating a bug, in most cases, we do not know the source of the bug for sure. Figure 10.2 shows an overview of how to apply our approaches in a reconfigurable system taking into account the challenge to be solved. Furthermore, if the approach does

	Design model	Model at runtime	Reconfigurations
EBRo	M	-	M
BLiMEA	M	-	-
DFL	M	M	-

Table 10.1: Mandatory and non-mandatory model elements necessary to apply our approaches

not perform well, we list some of the issues that we have found throughout the evaluations.

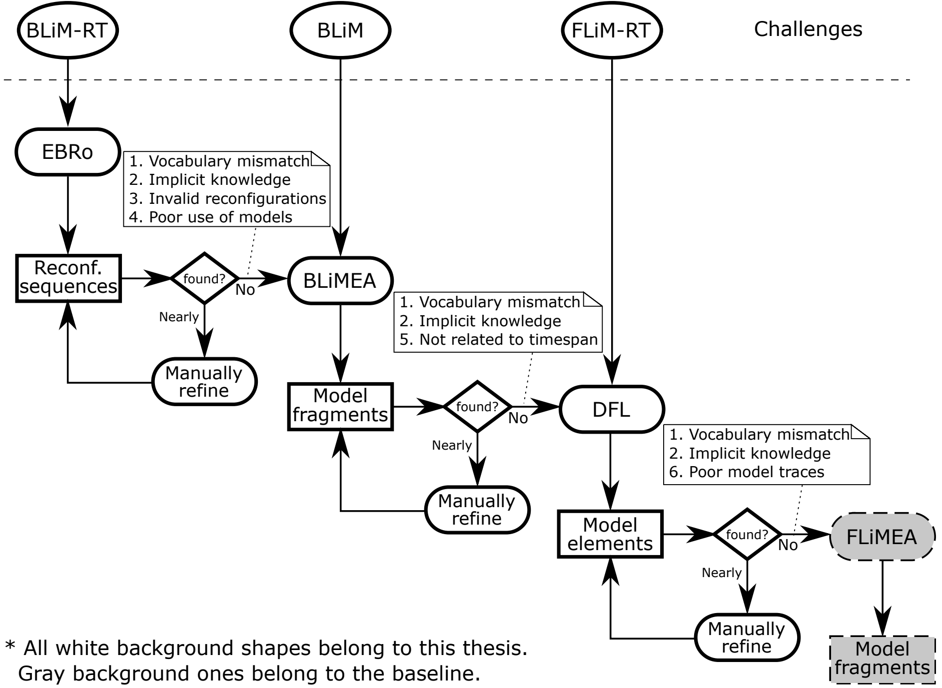


Figure 10.2: Application flow for a model-based reconfigurable system

The top part of Figure 10.2 presents the challenges: BLiM-RT, BLiM, and FLiM-RT. They are ordered from more specific to more general. For each of the challenges, we recommend starting by using one of our approaches: EBRo, BLiMEA, or DFL. However, some issues can arise (see notes of Figure 10.2).

Hence, we recommend applying specific approaches taking into account the problem to be solved and the system where the feature or bug is located. If the

results are not satisfactory, then we can manually refine the solutions or apply other approaches. In the example of Figure 10.2, we want to locate a bug and we have a system with dynamic reconfiguration rules. Our recommendation is to use the EBRo approach as the starting point. If the results obtained by EBRo are not useful, other approaches can be launched since they explore a larger solution space than EBRo. We recommend starting with EBRo because, if you find the bug, you would get more accurate information about it, i.e., the reconfiguration sequence that generates the bug. This idea of changing approaches arises as a result of jointly contemplating the approaches in this thesis. However, it still needs to be evaluated.

10.2 Issues detected in our approaches

This section presents the issues detected throughout our evaluations. These issues are: vocabulary mismatch, implicit knowledge, invalid reconfigurations, poor use of models, bugs that are not related to timespan modifications, and poor model traces.

10.2.1 Vocabulary mismatch

All of our approaches suffer from vocabulary mismatch. This means that for a specific concept, the terms used in the bug description are different from the terms used in the models. For example, the bug description includes the word *'current'* to refer to the electrical energy that reaches an inductor. However, in the models, the word *'power'* stands for the same concept. Nevertheless, this issue could be solved by augmenting the natural language processing (NLP) with a dictionary of synonyms.

In the same way, we have also detected cases in which in-house terms are used; for example, instead of using the word *'inverter'*, the name of a manufacturer is used (*'Fairchild'*). Therefore, the regular dictionary of synonyms would not work in this case. This suggests that the dictionary of synonyms should be refined by domain engineers to include in-home terms.

10.2.2 Implicit knowledge

Another issue is the case in which the bug description is incomplete. For example, in a bug description the following sentence can appear: *'The induction hob*

crashes when the user selects the power level 9 for a double inductor'. The engineers understand that the inductors have to reach that power level when the user selects it. However, this sentence also embodies implicit knowledge that is not written but is obvious to the domain engineers: *'The double inductor is formed by two concentric inductors'*, and *'If the pot that is on the top is large enough, both inductors have to reach the power level 9; otherwise, only the central inductor must reach that level.'*

Omitting words in the bug description negatively influences the fitness value of textual similarity since the fitness value of textual similarity is based on the co-occurrence of terms. This suggests that we must make the engineers aware of this issue. They should know that in cases in which the results obtained do not have enough quality and more model elements need to be located, they can reformulate the descriptions of the bugs in order to make the implicit knowledge explicit.

10.2.3 Invalid reconfiguration sequences

This issue is a specific problem of the EBRo approach because it is the only solution that uses the reconfiguration rules. We realized that some solutions are invalid sequences of reconfigurations that, theoretically, are not going to take place since the necessary context changes can never occur. In our future work, we will study the introduction of crossover and mutation operators that, by construction, do not generate invalid sequences. We will also study a repair operation for invalid sequences.

Although we are interested in the influence of narrowing the search space, we believe that we must maintain the possibility of generating sequences of theoretically invalid reconfigurations. For instance, induction hobs are sold all over the world and sometimes they are used in unforeseen ways, causing extremely unlikely sequences of reconfigurations to occur. These otherwise impossible reconfigurations are potential sources of bugs.

10.2.4 Poor use of models

We realized that the model elements that contained few attributes and methods got worse positions in the rankings than the ones that contained more attributes and methods. The information required by these elements was not as detailed as the other model elements when specifying the model. For this reason, these model elements got a lower position in the ranking.

In addition, the results suggest that there is a relationship between the use of models at runtime and the quality of the solutions. The results are better the more you have specified the reconfigurations using the models at runtime. In other words, the fewer the flow control operators (e.g., ifs) and auxiliary variables used to reconfigure the system, the better.

This finding suggests that the models at runtime pay off when doing bug localization. However, the finding must be taken cautiously, since our evaluation was not designed to verify the influence on the quality of the solutions of models at runtime, but rather to compare the performance of different approaches in the location of bugs in the reconfigurations of models at runtime.

10.2.5 Bugs not related to timespan modifications

Our results confirm the relevance of the Defect Localization Principle to models. The majority of bugs provided by the industrial partners (about 90%) are related to recent modifications. For bugs that are not related to recent modifications, our approach (in spite of including a timespan objective) obtains similar or slightly worse results than the baselines used in terms of recall, precision, and MCC.

This suggests that, given a bug description where we do not know whether or not the recent modification timespans are relevant, the inclusion of the objective of the Defect Localization Principle generally leads to better results than if it is not included. However, if the bugs are not related to recent modifications, it is not necessary to use an approach that takes into account the timespan to the last modification.

10.2.6 Poor model traces

Similarly to the poor use of models at runtime, model traces that contain more details about the runtime information, on average, obtain better results. However, they do not always get the best results. These traces are composed of more models than the model traces that contain less runtime information. In addition, two consecutive models of a trace that contains more runtime information are typically more similar than two consecutive models of a trace that contains less runtime information.

More information implies a larger search space in which we have to locate the feature. The fact that the models in the trace are similar can imply similarity of terms in the documents of our information retrieval technique (LSI). This may

cause the technique does not discriminate between some models. However, in 75% of the cases, the dynamic feature localization with model traces that contain more details about the runtime information obtains better results than the ones that contain less information.

10.3 Findings detected in our approaches

The following subsections present the findings detected throughout our evaluations. These findings are: reduction of search space, high level of abstraction, good performance between information retrieval and the defect localization principle, and non-reliance on the domain.

10.3.1 Reduction of search space

The goal of a feature or bug localization technique is to reduce the effort required by software engineers to find the desired feature or bug. We realize that the more specific the solution, the smaller the search space.

In addition, our model-based approaches reduce the search space compared to program-code-based approaches. Our model-based approaches, on average, require searching in fewer model elements than the program-code-based approaches.

For example, the Smart Hotel case study presents 68 model elements in its architecture model. The software components of the Smart Hotel consist of 268 classes that are implemented in about 67,207 methods of Java source code. When we applied our dynamic feature localization approach, the traces generated were, on average, composed of 46 model elements with our dynamic feature localization approach and 3,817 methods with the source-code based approach. Therefore, the reduction of search space is significant; searching in 46 elements does not require the same effort as searching in 3,817 elements.

10.3.2 High level of abstraction

Since models allow working at a high level of abstraction, the words used at the model level are closer to textual descriptions than the ones used in program-code-based approaches. The result is that queries using a natural language show better results with the model-based approaches. In program-code-based approaches, some general programming terms are taken into account. Some terms, like *'controller'* or *'run'*, originate from some programming patterns. By raising the level

of abstraction with the models, we can prevent general programming methods and variables from interfering with the feature location.

This leads to better discrimination in our approaches. The majority of the elements that are relevant to the feature or bug achieve better results than the ones that are not relevant. However, in the program-code-based approaches, the program-code methods that are relevant to the feature or bug and the ones that are not relevant are not differentiated.

10.3.3 Good performance between information retrieval and the defect localization principle

Bugs cannot be located using only textual similarity (as the baseline does) due to vocabulary mismatch and to descriptions with implicit knowledge. The reason is that some text is missing or that the texts of the description of the bug and the models are different.

Bugs cannot be located using only the defect localization principle. The reason is that all recent modifications would be suggested as being relevant to the bug. Therefore, the combination of textual similarity and the defect localization principle helps to locate bugs.

10.3.4 Non-reliance on the domain

Finally, we have applied our approaches to three case studies from very different domains. Two of them are real industrial cases (BSH and CAF). In all of the scenarios, we obtained good quality results. Hence, the findings suggest that our approaches do not rely on the domain because the results depend on the models themselves.

10.4 Comparison to other works

In recent years, many feature and bug localization approaches have been proposed. Table 10.2 shows a summary of some of the works and the scope of the approaches presented in them.

The first row of Table 10.2 includes approaches that only take into account the source code as the artifact that represents the feature or the bug. DiffJ [67] is a tool for comparing different versions of programs to find bugs. AML [70] and NetML [71] utilize multi-modal information from both bug reports and program spectra

	Scope		
	Design model	Model at runtime	Reconfigurations
Program-code approaches: DiffJ [67], AML [70] NetML [71], Amalgam+ [72]	×	×	×
Model-based approaches: MoVa2PL [100], BUT4Reuse [101], FLiMEA [102, 99]	✓	×	×
This thesis	✓	✓	✓

Table 10.2: Comparison to other approaches (✓= applied to the design model, the model at runtime, or reconfigurations; × = not applied).

to localize bugs. Amalgam+ [72] is a method for locating relevant buggy files that puts together fives sources of information, namely, version history, similar reports, structure, stack traces, and reporter information.

These approaches have not been applied to models. In contrast, our approaches apply the ideas used in source code (static and dynamic analysis, information retrieval, and the Defect Localization Principle) in models. We have evaluated the approaches successfully by applying them in models.

The second row of Table 10.2 includes some works that focus on the localization of features in models by comparing the models with each other in order to formalize the variability among them in the form of a Software Product Line. MoVa2PL [100] considers the identification of variability and commonality in model variants as well as the extraction of a CVL-compliant Model-based Software Product Line (MSPL) from the features identified on these variants. BUT4Reuse [101] is a bottom-up approach for implementing systematic software reuse.

Nevertheless, many of the model-based approaches are based on mechanical comparisons among the models, classifying the elements based on their similarity and identifying the dissimilar elements as feature realizations. In contrast, our work does not rely on model comparisons to locate the features or the bugs.

In addition, FLiMEA [102, 99] uses a genetic algorithm for feature localiza-

tion in models that could potentially be applied to perform both feature and bug localization even though they were not designed to locate bugs.

11

CONCLUSION

Contents

11.1 Overview 102
11.2 Research Questions 102
11.3 Ongoing Research 103
11.4 Related publications 105

11.1 Overview

This chapter recapitulates the results presented and concludes the dissertation. First, we answer the three research questions proposed in the dissertation. Then, the ongoing research is described. Finally, we conclude the dissertation.

11.2 Research Questions

The three research questions of the dissertation have been addressed through each of the evaluations presented. We present the answers to these research questions below:

Research Question 1: How does the use of runtime traces gives valuable additional information on only static program/models in Feature Location?

Answer to RQ1: To address RQ1, we developed an approach for feature localization in model traces. To do this, we combine models at runtime and Information Retrieval for feature location. The information is collected in the model at runtime and is filtered with information retrieval in order to extract the model elements that are relevant to the feature. The results show that the use of runtime information pays off when pursuing feature localization. Specifically, according to the experiments, the use of model traces outperforms the use of source code traces and the information recorded in the model traces influences the results.

Research Question 2: To what extent are the techniques for bug localization that are used in program code applicable to software based on models as well?

Answer to RQ2: To address RQ2, we developed an approach for bug localization in models (BLiMEA). To do this, the approach uses a Multi-objective Evolutionary Algorithm (MOEA) with two fitness functions: (1) Information Retrieval (IR); and (2) modification timespan. These two fitness functions are widely used in program-code-based approaches for bug localization. Hence, we adapted them to work with models. The results shows that the use of BLiMEA has an actual impact on bug localization in model-based systems, providing more accurate results than the approaches of the literature.

Research Question 3: How can we best locate bugs in model-based systems that are subject to dynamic reconfigurations?

Answer to RQ3: To address RQ3, we developed an approach that focuses on locating bugs that appear as the result of dynamic reconfigurations of the systems due to context changes (EBRo). To do this, the approach uses an evolutionary algorithm that is guided by a fitness function that considers the similarity to the description of the bug. The results show that there is empirical evidence to support the significance of the results of our algorithm. In addition, the solutions obtained with the EBRo approach are better than the ones obtained with other approaches.

11.3 Ongoing Research

The contributions presented in this dissertation are the results of the ongoing work that is currently being developed further. Specifically, the evolutionary algorithm used is currently being adapted to work under different conditions and applied to serve different purposes. This section presents some open research questions and the ongoing work that is being done to address them.

Parameter values of the evolutionary algorithm: Some of the approaches presented in this dissertation take advantage of evolutionary algorithms. Algorithms of this kind have several different parameters that can be modified and that can determine whether the search is successful or fails [103]. The problem of setting the parameters before running the algorithm is known as parameter tuning [103].

Throughout the work developed for this dissertation, we performed parameter tuning to find the best values for the parameters of our algorithms. However, the focus of this work is to compare the performance of the algorithms in terms of solution quality. We tried different values of the initial population and the crossover and mutation probability. Nevertheless, it did not have effect on our main focus. Even though we decided to maintain the default values commonly used in literature, determining which quantitative parameters provide the best results for each domain remains an open question and we are currently doing research to address this.

Good parameters in our algorithms for feature and bug localization can improve the results obtained. The solution of a specific problem such as our

feature or bug localization approaches requires a set of specific parameters that guarantee a satisfactory performance. The optimization of these parameters is a task that consumes a great amount of time and that often demands the subjective intervention of an expert.

Machine learning fitness: Our approaches use different fitness functions to explore the search space in which we want to locate the feature or the bug. Specifically, we have adapted functions that work in program code approaches to work with models.

Similarly, machine learning could be applied as a fitness function in our approaches. We are currently working on adapting our approaches to use machine learning [104]. One of the challenges of applying this technique in models is identifying the set of elements from a model that are relevant to the problem and encoding them into vectors.

We believe that the approaches presented in this dissertation can be upgraded with machine learning as a fitness function. This will allow our feature and bug localization approaches to search for patterns instead of textual similarity or timespan weightings. Some machine learning techniques have been applied for information retrieval with varying degrees of success [105].

Experiments with generic modeling languages We applied our approaches to domains with domain specific languages (DSLs). Unlike generic modeling languages such as UML, metamodel elements in DSLs contain domain information. On the one hand, the metaelement *Class* of UML is generic enough to be relevant for different domains. On the other hand, a metaelement such us *Inductor* of our Induction Hob DSL is relevant for specific domains only. This is also the case of other generic modeling languages, such as BPMN or ARCHIMATE.

This suggests that in the hypothetical case that our domains were specified with a generic modeling language, the results could have been worse since the terms of the model would not be similar to the terms of the description of the features or bugs. Therefore, we think that specific experiments should be carried out with generic modeling languages in order to determine the performance with the current approach, assess the need to reformulate the query, or even identify new objectives to guide the feature or bug localization.

Although our approaches have been applied to several domains based on MOF [106] with good results, we think that it is important to have more generalizable approaches. We plan to perform more evaluations with other modeling languages.

11.4 Related publications

Although there are some open research questions, the work presented in this dissertation has provided a step forward in terms of addressing the issues of feature and bug localization in model-based systems at design time and runtime. Specifically, the work presented in this dissertation includes the following:

Conferences: Our work has been presented at scientific venues (such as the 10th and 11th International Workshop on Models@run.time, the 9th International Conference on System Analysis and Modeling, the 13th European Conference on Modelling Foundations and Applications, the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, the 26th International Conference on Information Systems Development, and the ACM/IEEE 21th International Conference on Model Driven Engineering Languages and Systems).

Journals: Our work has been published in an international journal (namely Software & Systems Modeling [12]).

Research projects: Our work has contributed to national and international research projects such as VARIAMOS (Spanish national research project) and REVaMP2 (an international ITEA 3 Call 2 project).

Industrial scenarios: Our work has been evaluated in industrial scenarios such as BSH (the leading manufacturer of home appliances in Europe) and CAF (a worldwide provider of railway solutions).

BIBLIOGRAPHY

[1] Usman Mansoor, Marouane Kessentini, Philip Langer, Manuel Wimmer, Slim Bechikh, and Kalyanmoy Deb. Momm: Multi-objective model merging. *Journal of Systems and Software*, 103:423 – 439, 2015.

[2] Meir M Lehman, JF Ramil, and Goel Kahen. A paradigm for the behavioural modelling of software processes using system dynamics. Technical report, Imperial College of Science, Technology and Medicine, Department of Computing, Sep 2001.

[3] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Feature location in source code: A taxonomy and survey. In *Journal of Software Maintenance and Evolution: Research and Practice*, 2011.

[4] Julia Rubin and Marsha Chechik. A survey of feature location techniques. In Iris Reinhartz-Berger, Arnon Sturm, Tony Clark, Sholom Cohen, and Jorn Bettin, editors, *Domain Engineering*, pages 29–58. Springer Berlin Heidelberg, 2013.

[5] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, Aug 2016.

[6] Jaime Font Burdeus. *Location of Features as Model Fragments and their Co-Evolution*. PhD thesis, University of Oslo, Norway, 2017.

[7] Lorena Arcega, Jaime Font, Øystein Haugen, and Carlos Cetina. Leveraging models at run-time to retrieve information for feature location. In *Proceedings of the 10th International Workshop on Models@run.time co-located with the 18th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2015), Ottawa, Canada, September 29, 2015.*, pages 51–60, 2015.

[8] Lorena Arcega, Jaime Font, Øystein Haugen, and Carlos Cetina. An infrastructure for generating run-time model traces for maintenance tasks. In *Proceedings of the 11th International Workshop on Models@run.time co-located with the 19th International Conference on Model Driven Engineer-*

ing Languages and Systems (MoDELS 2016), Saint-Malo, France, October 4, 2016., 2016.

[9] Lorena Arcega, Jaime Font, Øystein Haugen, and Carlos Cetina. Feature location through the combination of run-time architecture models and information retrieval. In Jens Grabowski and Steffen Herbold, editors, *System Analysis and Modeling. Technology-Specific Aspects of Models : 9th International Conference, SAM 2016, Saint-Malo, France, October 3-4, 2016. Proceedings*, pages 180–195. Springer International Publishing, 2016.

[10] Lorena Arcega, Jaime Font, Øystein Haugen, and Carlos Cetina. On the influence of models at run-time traces in dynamic feature location. In *Modelling Foundations and Applications - 13th European Conference, ECMFA 2017, Held as Part of STAF 2017, Marburg, Germany, July 19-20, 2017, Proceedings*, 2017.

[11] Lorena Arcega, Jaime Font, Øystein Haugen, and Carlos Cetina. On the influence of modification timespan weightings in the location of bugs in models. In *Proceedings of the 26th International Conference on Information Systems Development, ISD 2017, Larnaca, Cyprus, September 6-8, 2017*, 2017.

[12] Lorena Arcega, Jaime Font, Øystein Haugen, and Carlos Cetina. An approach for bug localization in models using two levels: model and metamodel. *Software & Systems Modeling*, Mar 2019.

[13] Lorena Arcega, Jaime Font, and Carlos Cetina. Evolutionary algorithm for bug localization in the reconfigurations of models at runtime. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2018, Copenhagen, Denmark, October 14-19, 2018*, pages 90–100, 2018.

[14] Roelf J. Wieringa. *Design science methodology for information systems and software engineering*. Springer, Germany, 2014.

[15] Lorena Arcega, Jaime Font, Øystein Haugen, and Carlos Cetina. Achieving knowledge evolution in dynamic software product lines. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*, pages 505–516, 2016.

[16] Joaquin Miller, Jishnu Mukerji, et al. Mda guide version 1.0. 1, 2003.

[17] Francisco Durán Muñoz, Javier Troya Castilla, and Antonio Vallecillo Moreno. Desarrollo de software dirigido por modelos. *FUOC. Fundación para la Universitat Oberta de Catalunya*, 2007.

[18] Stuart Kent. Model driven engineering. In Michael Butler, Luigia Petre, and Kaisa Sere, editors, *Integrated Formal Methods*, pages 286–298, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[19] Jean-Marie Favre. Megamodelling and etymology. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2006.

[20] Frédéric Fondement and Raul Silaghi. Defining model driven engineering processes. In *Third International Workshop in Software Model Engineering (WiSME), held at the 7th International Conference on the Unified Modeling Language (UML)*, 2004.

[21] Aditya Agrawal, Tihamer Levendovszky, Jon Sprinkle, Feng Shi, and Gabor Karsai. Generative programming via graph transformations in the model-driven architecture. In *OOPSLA 2002 Workshop in Generative Techniques in the context of Model Driven Architecture*, 2002.

[22] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. volume 35, pages 26–36, New York, NY, USA, June 2000. ACM.

[23] Juha-Pekka Tolvanen and Steven Kelly. Modelling languages for product families: A method engineering approach. In *Proceedings of OOPSLA workshop on Domain-Specific Visual Languages*, 2001.

[24] Mark A. Simos. *Organization Domain Modeling (ODM): Formalizing the Core Domain Modeling Life Cycle*. SSR '95. ACM, New York, NY, USA, 1995.

[25] Robert Esser and Jörn W. Janneck. A framework for defining domain-specific visual languages. In *OOPSLA 2001 Workshop on Domain Specific Visual Languages*, 2001.

[26] Nelly Bencomo, Robert France, Betty H. C. Cheng, and Uwe Aßmann, editors. *Models@run.time. Foundations, Applications, and Roadmaps.* Springer International Publishing, 2014.

[27] IBM. An architectural blueprint for autonomic computing. Technical report, IBM, 2006.

[28] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016.

[29] J. O. Kephart and W. E. Walsh. An artificial intelligence perspective on autonomic computing policies. In *Proceedings of the Fifth IEEE International Workshop on Policies for Distributed Systems and Networks. POLICY 2004*, pages 3–12, June 2004.

[30] K. Chen and V. Rajlich. Case study of feature location using dependence graph. In *Proceedings IWPC 2000. 8th International Workshop on Program Comprehension*, pages 241–247, June 2000.

[31] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, 29(3):210–224, March 2003.

[32] M. Eaddy, A.V. Aho, G. Antoniol, and Y.-G. Gueheneuc. Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pages 53–62, June 2008.

[33] Rainer Koschke and Jochen Quante. On dynamic feature location. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 86–95, New York, NY, USA, 2005. ACM.

[34] Dapeng Liu, Andrian Marcus, Denys Poshyvanyk, and Vaclav Rajlich. Feature location via information retrieval based filtering of a single scenario execution trace. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 234–243, New York, NY, USA, 2007. ACM.

[35] A. Marcus, A. Sergeyev, V. Rajlich, and J.I. Maletic. An information retrieval approach to concept location in source code. In *Proceedings of the 11th Working Conference on Reverse Engineering*, pages 214–223, Nov 2004.

[36] M. Revelle, B. Dit, and D. Poshyvanyk. Using data fusion and web mining to support feature location in software. In *IEEE 18th International Conference on Program Comprehension (ICPC)*, pages 14–23, June 2010.

[37] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. Reverse engineering feature models. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 461–470, New York, NY, USA, 2011. ACM.

[38] A.D. Eisenberg and K. De Volder. Dynamic feature traces: finding features in unfamiliar code. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 337–346, Sept 2005.

[39] Norman Wilde and Michael C. Scully. Software reconnaissance: Mapping program features to code. *Journal of Software Maintenance*, 7(1):49–62, January 1995.

[40] W. E. Wong, S. S. Gokhale, J. R. Horgan, and K. S. Trivedi. Locating program features using execution slices. In *Proceedings 1999 IEEE Symposium on Application-Specific Systems and Software Engineering and Technology. ASSET'99 (Cat. No.PR00122)*, pages 194–203, March 1999.

[41] Denys Poshyvanyk, Yann-Gael Gueheneuc, Andrian Marcus, Giuliano Antoniol, and Vaclav Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, 33(6):420–432, June 2007.

[42] S. Duszynski, J. Knodel, and M. Becker. Analyzing the source code of multiple software variants for reuse potential. In *2011 18th Working Conference on Reverse Engineering*, pages 303–307, Oct 2011.

[43] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 805–824, New York, NY, USA, 2011. ACM.

[44] Christian Kästner, Klaus Ostermann, and Sebastian Erdweg. A variability-aware module system. In *Proceedings of the ACM International Confer-*

ence on Object Oriented Programming Systems Languages and Applications, OOPSLA '12, pages 773–792, New York, NY, USA, 2012. ACM.

[45] C. Kästner, A. Dreiling, and K. Ostermann. Variability mining: Consistent semi-automatic detection of product-line features. *IEEE Transactions on Software Engineering*, 40(1):67–82, Jan 2014.

[46] V. Alves, C. Schwanninger, L. Barbosa, A. Rashid, P. Sawyer, P. Rayson, C. Pohl, and A. Rummler. An exploratory study of information retrieval techniques in domain analysis. In *2008 12th International Software Product Line Conference*, pages 67–76, Sept 2008.

[47] K. Czarnecki and A. Wasowski. Feature diagrams and logics: There and back again. In *11th International Software Product Line Conference (SPLC 2007)*, pages 23–34, Sep. 2007.

[48] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. Mining configuration constraints: Static analyses and empirical results. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 140–151, New York, NY, USA, 2014. ACM.

[49] N. Niu and S. Easterbrook. On-demand cluster analysis for product line functional requirements. In *2008 12th International Software Product Line Conference*, pages 87–96, Sep. 2008.

[50] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., New York, NY, USA, 1986.

[51] Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407, 1990.

[52] David M. Blei, Andrew Y. Ng, Michael I. Jordan, and John Lafferty. Latent dirichlet allocation. *Journal of Machine Learning Research*, 3(4/5):993 – 1022, 2003.

[53] Stephen W. Thomas, Ahmed E. Hassan, and Dorothea Blostein. *Mining Unstructured Software Repositories*, pages 139–162. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[54] I. Chawla and S. K. Singh. Performance evaluation of vsm and lsi models to determine bug reports similarity. In *2013 Sixth International Conference on Contemporary Computing (IC3)*, pages 375–380, Aug 2013.

[55] Md Masudur Rahman, Saikat Chakraborty, and Baishakhi Ray. Which similarity metric to use for software documents?: A study on information retrieval based software engineering tasks. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, pages 335–336, New York, NY, USA, 2018. ACM.

[56] F. Asadi, M. Di Penta, G. Antoniol, and Y. Gueheneuc. A heuristic-based approach to identify concepts in execution traces. In *2010 14th European Conference on Software Maintenance and Reengineering*, pages 31–40, March 2010.

[57] Bogdan Dit, Meghan Revelle, and Denys Poshyvanyk. Integrating information retrieval, execution and link analysis algorithms to improve feature location in software. *Empirical Software Engineering*, 18(2):277–309, 2013.

[58] Nelly Bencomo, Sebastian Götz, and Hui Song. Models@run.time: a guided tour of the state of the art and research challenges. *Software & Systems Modeling*, Jan 2019.

[59] Samuel Kounev, Jeffrey O. Kephart, Aleksandar Milenkoski, and Xiaoyun Zhu. *Self-Aware Computing Systems*. Springer Publishing Company, Incorporated, 1st edition, 2017.

[60] Shivani Rao and Avinash Kak. Retrieval from software libraries for bug localization: A comparative study of generic and composite text models. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 43–52, New York, NY, USA, 2011. ACM.

[61] B. Sisman and A. C. Kak. Incorporating version histories in information retrieval based bug localization. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 50–59, June 2012.

[62] Jian Zhou, Hongyu Zhang, and David Lo. Where should the bugs be fixed? - more accurate information retrieval-based bug localization based on bug reports. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 14–24, Piscataway, NJ, USA, 2012. IEEE Press.

[63] Dongsun Kim, Yida Tao, Sunghun Kim, and Andreas Zeller. Where should we fix this bug? a two-phase recommendation model. *IEEE Transactions on Software Engineering*, 39(11):1597–1610, 2013.

[64] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry. Improving bug localization using structured information retrieval. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 345–355, Nov 2013.

[65] L. Burgueño, J. Troya, M. Wimmer, and A. Vallecillo. Static fault localization in model transformations. *IEEE Transactions on Software Engineering*, 41(5):490–506, May 2015.

[66] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. Bug localization with combination of deep learning and information retrieval. In *Proceedings of the 25th International Conference on Program Comprehension, ICPC 2017, Buenos Aires, Argentina, May 22-23, 2017*, pages 218–229, 2017.

[67] Elton Alves, Milos Gligoric, Vilas Jagannath, and Marcelo d'Amorim. Fault-localization using dynamic slicing and change impact analysis. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 520–523, Washington, DC, USA, 2011. IEEE Computer Society.

[68] L. Gong, D. Lo, L. Jiang, and H. Zhang. Interactive fault localization leveraging simple user feedback. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 67–76, Sept 2012.

[69] Xiaoguang Mao, Yan Lei, Ziying Dai, Yuhua Qi, and Chengsong Wang. Slice-based statistical fault localization. *J. Syst. Softw.*, 89:51–62, March 2014.

[70] Tien-Duy B. Le, Richard J. Oentaryo, and David Lo. Information retrieval and spectrum based bug localization: Better together. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 579–590, New York, NY, USA, 2015. ACM.

[71] Thong Hoang, Richard Jayadi Oentaryo, Tien-Duy B. Le, and David Lo. Network-clustered multi-modal bug localization. *CoRR*, abs/1802.09729, 2018.

[72] Shaowei Wang and David Lo. Amalgam+: Composing rich information sources for accurate bug localization. *Journal of Software: Evolution and Process*, 28(10):921–942, 2016.

[73] Ilhem Boussaïd, Patrick Siarry, and Mohamed Ahmed-Nacer. A survey on search-based model-driven engineering. *Automated Software Engineering*, 24(2):233–294, Jun 2017.

[74] Øystein Haugen, Birger Møller-Pedersen, Jon Oldevik, G?ran K. Olsen, and Andreas Svendsen. Adding standardized variability to domain specific languages. In *Proceedings of the 2008 12th International Software Product Line Conference*, SPLC '08, pages 139–148, Washington, DC, USA, 2008. IEEE Computer Society.

[75] Andreas Svendsen, Xiaorui Zhang, Roy Lind-Tviberg, Franck Fleurey, Øystein Haugen, Birger Møller-Pedersen, and G?ran K. Olsen. Developing a software product line for train control: A case study of cvl. In *Proceedings of the 14th International Conference on Software Product Lines: Going Beyond*, SPLC'10, pages 106–120, Berlin, Heidelberg, 2010. Springer-Verlag.

[76] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Commun. ACM*, 18(11):613–620, November 1975.

[77] Thomas K Landauer, Peter W. Foltz, and Darrell Laham. An introduction to latent semantic analysis. *Discourse Processes*, 25(2-3):259–284, 1998.

[78] Anette Hulth. Improved automatic keyword extraction given more linguistic knowledge. In *Proceedings of the 2003 Conference on Empirical Methods in Natural Language Processing*, EMNLP '03, pages 216–223, Stroudsburg, PA, USA, 2003. Association for Computational Linguistics.

[79] Hamzeh Eyal Salman, Abdelhak Seriai, and Christophe Dony. 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.*, pages 426–430, 2014.

[80] A. E. Hassan and R. C. Holt. The top ten list: dynamic fault prediction. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 263–272, Sept 2005.

[81] Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 563–572, Washington, DC, USA, 2004. IEEE Computer Society.

[82] Zbigniew Michalewicz. *Genetic algorithms + data structures = evolution programs*. Artificial intelligence. Springer, 1992.

[83] Michael Affenzeller, Stephan M. Winkler, Stefan Wagner, and Andreas Beham. *Genetic Algorithms and Genetic Programming - Modern Concepts and Practical Applications*. CRC Press, 2009.

[84] Sima Zamani, Sai Peck Lee, Ramin Shokripour, and John Anvik. A noun-based approach to feature location using time-aware term-weighting. *Information and Software Technology*, 56(8):991 – 1011, 2014.

[85] Huzefa Kagdi, Malcom Gethers, Denys Poshyvanyk, and Maen Hammad. Assigning change requests to software developers. *Journal of Software: Evolution and Process*, 24(1):3–33, 2012.

[86] James E. Baker. Reducing bias and inefficiency in the selection algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms on Genetic Algorithms and Their Application*, pages 14–21. L. Erlbaum Associates Inc., 1987.

[87] Carlos Cetina. *Achieving Autonomic Computing through the Use of Variability Models at Run-time*. PhD thesis, Universidad Politécnica de Valencia, 2010.

[88] Javier Muñoz. *Model Driven Development of Pervasive Systems. Building a Software Factory*. PhD thesis, Universidad Politécnica de Valencia, 2008.

[89] N. Bencomo, S. Hallsteinsen, and E. Santana de Almeida. A view of the dynamic software product line landscape. *Computer*, 45(10):36–41, Oct 2012.

[90] Stephen V. Stehman. Selecting and interpreting measures of thematic classification accuracy. *Remote Sensing of Environment*, 62(1):77 – 89, 1997.

[91] Sabri Boughorbel, Fethi Jarray, and Mohammed El-Anbari. Optimal classifier for imbalanced data using matthews correlation coefficient metric. *PLOS ONE*, 12(6):1–17, 06 2017.

[92] D. M. W. Powers. Evaluation: From precision, recall and f-measure to roc., informedness, markedness & correlation. *Journal of Machine Learning Technologies*, 2(1):37–63, 2011.

[93] Andrea Arcuri and Lionel Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Softw. Test. Verif. Reliab.*, 24(3):219–250, May 2014.

[94] Salvador García, Alberto Fernández, Julián Luengo, and Francisco Herrera. Advanced nonparametric tests for multiple comparisons in the design of experiments in computational intelligence and data mining: Experimental analysis of power. *Information Sciences*, 180(10):2044 – 2064, 2010. Special Issue on Intelligent Distributed Information Systems.

[95] W. J Conover. *Practical Nonparametric Statistics, 3rd Edition*. Wiley, 1999.

[96] Andrea Arcuri and Gordon Fraser. Parameter tuning or default values? an empirical investigation in search-based software engineering. *Empirical Software Engineering*, 18(3):594–623, 2013.

[97] András Vargha and Harold D. Delaney. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

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

[99] Jaime Font, Lorena Arcega, Øystein Haugen, and Carlos Cetina. Feature location in models through a genetic algorithm driven by information retrieval techniques. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, MODELS '16, pages 272–282. ACM, 2016.

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

[101] Jabier Martinez, Tewfik Ziadi, Tegawendé F. Bissyandé, and Yves Le Traon. Bottom-up adoption of software product lines: A generic and

extensible approach. In *Proceedings of the 19th International Software Product Line Conference*, SPLC '15, Nashville, TN, USA., 2015.

[102] Jaime Font, Lorena Arcega, Øystein Haugen, and Carlos Cetina. Feature location in model-based software product lines through a genetic algorithm. In *15th International Conference on Software Reuse*, ICSR 2016, Limassol, Cyprus, Jun 2016.

[103] A. E. Eiben and S. K. Smit. *Evolutionary Algorithm Parameters and Methods to Tune Them*, pages 15–36. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[104] Ana Cristina Marcén, Francisca Pérez, and Carlos Cetina. Ontological evolutionary encoding to bridge machine learning and conceptual models: Approach and industrial evaluation. In *Conceptual Modeling - 36th International Conference, ER 2017, Valencia, Spain, November 6-9, 2017, Proceedings*, 2017.

[105] W. Bruce Croft. Machine learning and information retrieval. In Armand Prieditis and Stuart Russell, editors, *Machine Learning Proceedings 1995*, page 587. Morgan Kaufmann, San Francisco (CA), 1995.

[106] Meta object facility (mof) 2.0 core specification, 2003. Version 2.

Part V

PUBLICATIONS

12

Dynamic Feature Localization in Models

Contents

12.1 MRT’15 Paper 123
12.2 MRT’16 Paper 135
12.3 SAM’16 Paper 145
12.4 ECMFA’17 Paper 163
12.5 SANER’16 Paper 181

12.1 MRT'15 Paper

Title: Leveraging Models at Run-time to Retrieve Information for Feature Location.

Authors: Lorena Arcega, Jaime Font, Øystein Haugen, Carlos Cetina.

Proceedings: Proceedings of the 10th International Workshop on Models@run.time.

Location: Ottawa, Canada - September 29, 2015

Publisher: CEUR Workshop Proceedings

Pages: 51-60

DOI: `http://ceur-ws.org/Vol-1474`

Contribution: Lorena Arcega is the main author of the paper and is responsible for 90% of the work. She was also responsible for the oral presentation of the work which took place during the conference.

Leveraging Models at Run-time to Retrieve Information for Feature Location

Lorena Arcega[1,2], Jaime Font[1,2] Øystein Haugen[2,3], and Carlos Cetina[1]

[1] San Jorge University, SVIT Research Group, Zaragoza, Spain
{larcega,jfont,ccetina}@usj.es
[2] University of Oslo, Department of Informatics, Oslo, Norway
oysteinh@ifi.ui.no
[3] Østfold University College, Department of Information Technology, Halden, Norway
oystein.haugen@hiof.no

Abstract. Model Driven Engineering (MDE) has the potential to be used at run-time, to monitor and verify particular aspects of run-time behaviour. Models at run-time provide a kind of formal basis for reasoning about the current system state at run-time, for reasoning about necessary adaptations, and for analyzing or predicting the consequences of possible system adaptations. However, we believe that models at run-time paradigm can be useful in other research areas such as variability extraction and feature location. This work proposes the use of models at run-time for increasing the information for feature location. We have tried this work with a Smart Hotel defined with an architecture model at run-time and driven by a reconfiguration loop. The results indicate that the models at run-time paradigm generates information that can be used in the area of feature location. In addition, the results show that there is potential in combining these two research areas: models at run-time and feature location.

Keywords: Models@Run-time, Common Variability Language, Reverse engineering

1 Introduction

Model Driven Engineering (MDE) is used at run-time, to monitor and verify particular aspects of run-time behaviour [1]. Models at run-time provide a kind of formal basis for reasoning about the current system state, for reasoning about necessary adaptations, and for analyzing the consequences of possible system adaptations. Models at run-time development approaches have the proven capability to deliver complex, dependable software efficiently and effectively.

Other research areas are focused on variability extraction and feature location. Currently research efforts in feature location are concerned with identifying software artifacts (source code) associated with a program functionality (a feature). Feature location is one of the most important and common activities performed by developers during software maintenance and evolution [3].

In Models at run-time, the approaches define a causal connection between the system and the run-time model (there is a bidirectional relation between the source code and the run-time model). We believe that the information extracted from models at run-time approaches can be useful in the feature location field.

This work proposes the use of models at run-time for increasing the information for feature location. We develop an algorithm for retrieving the model information from a given feature selected by the software engineer (target feature). The input of the algorithm is an execution trace from a models at run-time approach. This trace contains all the possible configurations, as well as the number of times they occurred during the execution time. Then, the software engineer has to select a model fragment which he believes takes part of the target feature. Our algorithm considers this model fragment as a seed for the next step. This seed is mutated taking into account the models of the configurations extracted from the trace. The result is a ranking of model fragments that can be part of the target feature. In the last step, the software engineer has to manually select the model fragment that he believes is the one corresponding to the target feature taking into account the ranking information.

We have tried this work with a Smart Hotel defined with an architecture model at run-time and driven by a reconfiguration loop. In addition, we have the feature model corresponding to the architecture model. We use this feature model as an oracle to know the accuracy of the final results. The average of the comparisons indicates that the software engineer didn't select the correct fragment model for the target features. However, the fragment model corresponding to the target feature was in the top of the ranking. Although we have to tune the ranking information, the results of the evaluation indicate that the models at run-time paradigm generates useful information for the feature location.

Finally, we realize that the combination of models at run-time area with feature location area requires more research. Some of these open questions are related to the length of the trace that we need for more accurate information, and the information that we can extract from the transitions between configurations. Hence, the number of fragment mutations that is done by our algorithm needs refinements, as well as the restrictions that we can select in the mutations. Finally, this work should be compared with other dynamic techniques for feature location. Thus, this work could be used in combination with other feature location techniques for extending the information retrieved about the features.

The remainder of the paper is structured as follows. In Section 2, we present the motivation of this work. In Section 3, we present the Smart Hotel. In Section 4, we introduce our algorithm. In Section 5, we illustrate our algorithm with the Smart Hotel.In Section 6, we present the discussion of the results. In Section 7, we examine the related work, and we present the conclusions in Section 8.

2 Motivation

Reverse Variability engineering approaches are focused on variability extraction and feature location [3]. In feature location, some of the techniques include

dynamic analysis. Dynamic analysis refers to examining software system's execution. That is, feature location using dynamic analysis relies on a post-mortem analysis of an execution trace to find the source code of a specific feature.

Trace analysis is the main technique used at run-time to extract relevant information to create the variability model. When the system under study is executed, it generates a trace indicating which parts of the code have been executed. Usually, they compare the traces produced when a certain feature is executed with the traces produced when a certain feature is not executed to isolate the parts of the code involved in such feature. Some approaches ([12, 5]) are based solely on the trace analysis while other works combine dynamic analysis with other static analysis ([4, 11]).

Models at run-time approaches define a causal connection between the system and the run-time model. That is, there is a bidirectional relation between the source code and the run-time model. The benefit of using models at run-time is that they can provide richer traces taking advantage of the source code and the models.

Hence, we believe that the information extracted from models at run-time approaches can be useful in the feature location field. These traces that combine source code with run-time models can provide more data than the traces that only take into account the source code.

3 The Smart Hotel

The example of this paper is performed through a reconfigurable Smart Hotel [2]. The run-time reconfigurations are performed by an implementation of a MAPE-K loop [8] named Model-based Reconfiguration Engine (MoRE) [2]. In this section, we present MoRE reconfiguration MAPE steps and the Domain Specific Language (DSL) that MoRE uses as knowledge to switch between configurations of the Smart Hotel.

We use Pervasive Modelling Language (PervML) [10] to describe the Smart Hotel architecture. PervML is a DSL that describes pervasive systems using high-level abstraction concepts based on Meta-Object Facility (MOF) [1]. This language is focused on specifying heterogeneous services in specific physical environments such as the services of a Smart Hotel. This DSL has been applied to develop solutions in the Smart Hotel domain. The PervML language provides different models to specify the services and devices of a pervasive system.

Due to space constraints, in this work, we only focus on the subset of PervML that specifies the relationships among devices and services. This subset specifies the components that define a particular configuration system (services and devices) and how these components are connected with each other (channels). Services are depicted by circles, devices are depicted by squares, and the channels connecting services and devices are depicted by lines (see Fig. 1).

In MoRE, the Monitor (M) uses the run-time state as input to check context conditions. If any of these conditions are fulfilled, the Analyzer (A) uses the

[1] Meta object facility (MOF) 2.0 core specification, 2003

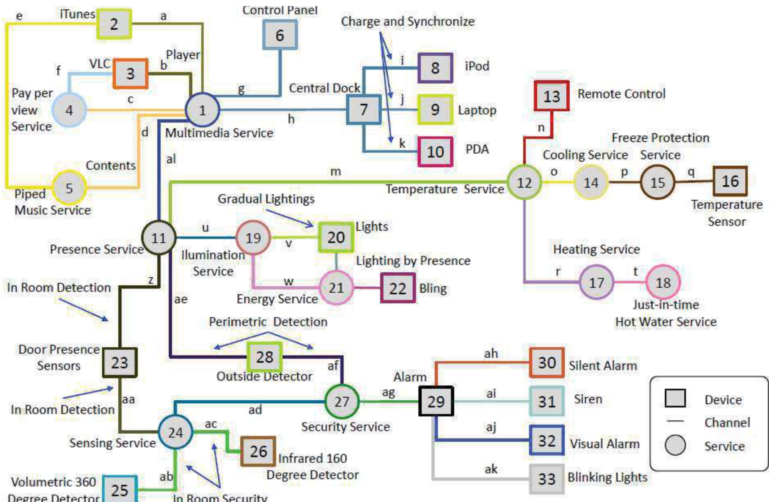


Fig. 1. Smart Hotel Architecture Model

associated resolution and the previous model operations to query the run-time models about the necessary modifications. The response of the models is used by the Planner (P) to elaborate a reconfiguration plan. This plan contains reconfiguration actions, which modify the system architecture and maintain the consistency between the models and the system architecture. The Execution (E) of this plan modifies the architecture by executing reconfiguration actions that deal with the activation and deactivation of components and the creation and destruction of channels among components.

MoRE calculates the architecture increments and decrements in order to determine the actions necessary to modify the system architecture. The adaptations policies of the Smart Hotel are expressed by means of optimizations algorithms that depend on the inputs at run-time. For this reason, the configurations of the Smart Hotel are not known at the beginning.

4 Feature location with Models at Run-time

Our algorithm for feature location is based on identification and extraction of model fragments related to a given feature. Fig. 2 presents an overview of the feature location algorithm to identify and extract model fragments, which consists of four steps.

The first step (Models@RT Traces) gets the input of the algorithm. It gets the trace resulting from a system that has been running for a specified time.

In second step (Fragments Mutation), the software engineer decides which feature to locate (target feature). The step performs automatic mutations of

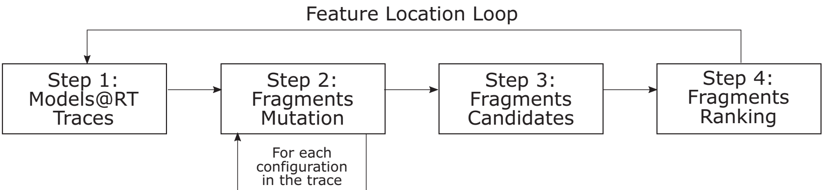


Fig. 2. Feature Location Algorithm

the model fragment designated as seed. The seed is selected by the software engineer, it is a model fragment of the complete architecture model that the engineer believes takes part of the target feature. The selection of the model fragment is based on the intuition of the software engineer of what parts of the model could be part of the target feature. The fragments are formalized by means of the Common Variability Language (CVL) [7]. This step takes as input the architecture model from the different configurations of the trace and the selected seed. The result is a set of fragments that are variations of the seed fragment.

The mutations are performed taking into account the model and the seed. Taking the seed fragment model as starting point, some model elements are added to or removed from the seed model fragment. The elements added during mutations are obtained from the corresponding architecture model from each configuration. The generated fragment is a subset of model elements from the corresponding architecture model of the configuration. Hence, we can guarantee that the generated fragments are part of the architecture model of each configuration. This step is performed as many times as different configurations in the trace to extract all possible fragments.

The third step (Fragments Candidates) assesses each fragment obtained in the second step. This step is performed automatically. The algorithm checks on how many occasions the model fragment appears in the trace. The values are assigned depending on the configurations in which the fragment appears and the number of times that these configurations appear in the trace. That is, each fragment has a point for each of the different configurations in which it appears, and each fragment has a point for each time the configuration in which it appears is present in the trace.

In the fourth step (Fragments Ranking), the last one, the fragments are ordered in a ranking taking into account the values obtained in the previous step. The ranking is composed by all the model fragments obtained from the different configurations. The model fragments with higher values are in the top part of the ranking because they are the most relevant to the target feature.

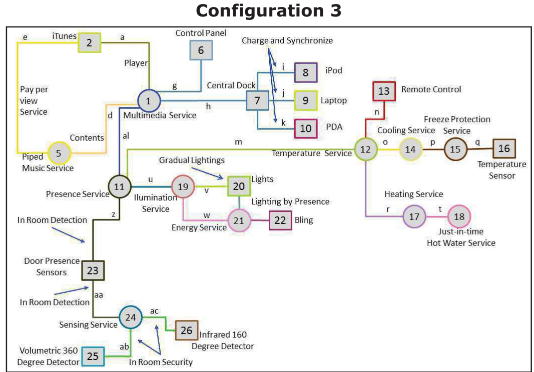
Taking into account this information, the software engineer can select the model fragment that best fits their understanding of the target feature. For instance, he can select the initial seed selection, however some of the model fragments can provide more relevant information for the feature.

The algorithm can be repeated until all the recognizable features of the architecture model have been located. The number of loops needed depends on the domain where it is being applied and the amount of variability that must be formalized.

5 Example: Feature Location in the Smart Hotel

We tried our algorithm with a Smart Hotel defined with an architecture model and driven by a reconfiguration loop (MoRE). The role of the software engineer was carried out by a master student outside this work.

Configuration 3



Configuration 3 Mutations - Seed: Multimedia service (1)

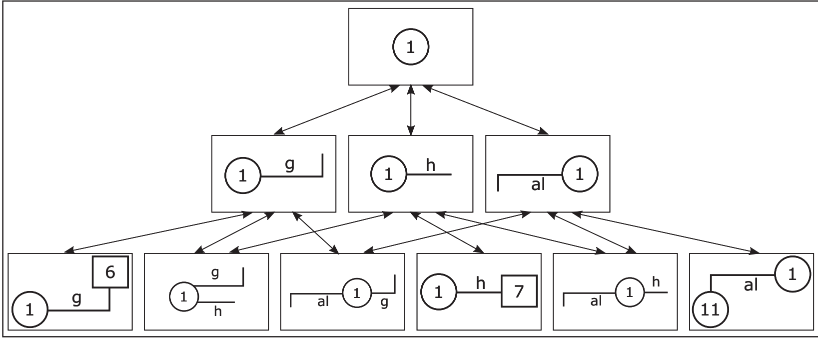


Fig. 3. Configuration 3 and Mutations of the Seed Fragment

We executed the Smart Hotel software during a time. The Smart Hotel reconfigured itself thirty times between twelve different configurations.

In this example, the feature that the software engineer wanted to locate is the feature related to Multimedia services. The software engineer selected the

Model fragments	1 g 6	1	...	11 al 1	1 h	...	1 d	2 a 1 g
Ocurrences in configurations	10/12	10/12	...	8/12	7/12	...	3/12	1/12
Ocurrences in reconfigurations	25/30	25/30	...	19/30	19/30	...	9/14	4/14

Fig. 4. Model Fragments Ranking

Multimedia Service (see circle 1 of Fig. 1) as seed because he believes it is the model fragment that best fits with the target feature.

The algorithm performed mutations taking into account each of the models (one at a time) and the selected seed. Fig. 3 shows the application of the step. The graph represents all the fragments obtained from the mutations. Each node (rectangle) represents a fragment. In this case, the image shows the mutations corresponding to the Configuration 3 architecture model of the Smart Hotel. There are three mutation levels, however the algorithm can be restricted to calculating fragments up to a fixed depth level. Top part of Fig. 3 shows the model fragment selected as seed (Multimedia Service). The rest of the graph contains possible fragments that can correspond to the target feature (Multimedia).

Fig. 4 shows the application of the four step of the algorithm. Each column shows each model fragment while each row shows the information about each one of the fragments. Second row (Occurrences in configurations) shows in how many configurations appears the fragment. In this case the numbers are x/12 because the trace contains twelve configurations. Third row (Occurrences in reconfigurations) shows how many times appears the configuration in the trace. In this case the numbers are x/30 because the trace contains thirty reconfigurations.

After the application of the algorithm the software engineer had to decide which is the model fragment that corresponds to the searching feature. Fig. 5 first column (Chosen by the user) shows the model fragment that he selected for the target feature Multimedia.

In this case, for checking the results we had the feature model that corresponds to the architecture model of Fig. 1. Then, we could use this feature model as an oracle to check the response of the software engineer. The oracle indicated that the fragment chosen was not the correct one for the feature Multimedia. Fig. 5 second and third columns (Oracle) shows the correct fragment for the feature Multimedia and the corresponding features corresponding to the software engineer choice.

Fig. 5 shows that the correct fragment for the feature was the one which was chosen as seed. The model fragment selected by the software engineer corresponds to the target feature and to an optional feature that can be selected once the target feature is selected (see Fig. 5 third column).

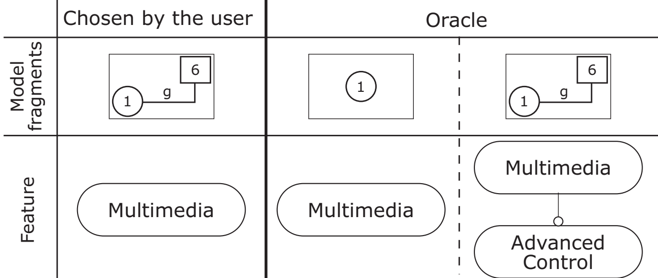


Fig. 5. Results of the Evaluation

6 Discussion

Although the user have not chosen the right model fragment for the feature Multimedia, Fig. 4 shows that the right model fragment has the same values than the model fragment selected in the ranking. Both appear in the top part of the ranking because the two fragments are relevant for the feature.

This work presents our ongoing work and our preliminary results of the application of our algorithm in a reconfigurable Smart Hotel. Despite the fact that the results shown are for locating only one feature, we have done tests for other features (i.e., Security, Automated Illumination,. . .) with similar results. The results and the evaluations performed indicate that we need to tune our algorithm to get accurate information.

Two of the main improvements for our algorithm could be the creation of an heuristic to determine how many mutation levels are needed, and the possibility of establishing restriction in the mutations of the fragments. The heuristic can help us to determine the number of levels needed in the mutations depending on the domain, the size of the architecture model and the accuracy needed. The restrictions allow us to limit the fragments that can appear in the mutations. For example, in the Smart Hotel we can restrict the mutations to fragments that connect one service with one device. The result will be the fragments that satisfy the restriction while the fragments that don't satisfy the restriction will be discarded.

In addition, future work can contribute to the feature location area providing data about the trace so that it contains enough information for feature location. Our example shows that the software engineer has not chosen the right model fragment. He selected the model fragment that corresponds to the target feature and to an optional feature that can be selected once the target feature is selected. This is because in all the configurations of the trace used in the algorithm the target feature and the optional feature are selected, hence both appear in the architecture model. This error could have been solved with a longer trace or other trace containing more relevant information.

Furthermore, leveraging models at run-time, some extra data from the transitions between configurations could improve the information that can be shown to the user to select the correct model fragment for the target feature.

Finally, we can obtain more information for improving our algorithm if we perform the same example of our work with other feature location techniques. Comparing the results may make us realize the weaknesses of our algorithm. Moreover, we can find some other technique that can cover these weak areas. Thus, this work could be used in combination with other feature location techniques for extending the information retrieved about the features.

7 Related work

To the best of our knowledge, there are no research efforts in the models at run-time area to locate features. Some approaches use design-time models to extract variability as follows.

Zhang et al. [14] present an approach to compare models to obtain the differences between them. This variability is used to build a variability model that is presented to the user to be validated and extended. Font et al. [6] propose to identify model patterns by human-in-the-loop and conceptualize them as reusable model fragments. Their approach provides the means to identify and extract those model patterns and further apply them to existing product models. Martinez et al. [9] propose an extensible framework that allows to identify, locate and extract features from the models. As a result, the task of adopting a software product line from a family of models reducing the initial investment required is provided. Wille et al. [13] present an approach to compare products from a family, focusing on the extraction of the variability between the interfaces of the different components in the models.

All of these works are based on extract model fragments from a given set of models. However, these approaches don't take into account the run-time behaviour of the systems.

8 Conclusion

This work extends the feature location techniques leveraging the models at run-time paradigm. Specifically, our algorithm retrieves information of the run-time models for improving feature location in reconfigurable systems.

Although we realize that the combination of models at run-time area and feature location area requires more research, our evaluation shows the preliminary results of how the models at run-time can generate useful information at model level.

In the near future, we would like to explore and improve the weak points of our algorithm: the number of fragment mutations that is necessary, and the restrictions that we can select in the mutations. In addition more research is necessary to exploit all the benefits that models at run-time can contribute to

the feature location: the information that we can extract from the transitions between configurations, and the length of the trace that we need for more accurate information. Finally, we can obtain more information for our algorithm if we compare our work with other feature location techniques.

References

1. Bencomo, N., France, R., Cheng, B.H.C., Amann, U. (eds.): Models@run.time. Foundations, Applications, and Roadmaps. Springer International Publishing (2014)
2. Cetina, C.: Achieving Autonomic Computing through the Use of Variability Models at Run-time. Ph.D. thesis, Universidad Politcnica de Valencia (2010)
3. Dit, B., Revelle, M., Gethers, M., Poshyvanyk, D.: Feature location in source code: A taxonomy and survey. In: Journal of Software Maintenance and Evolution: Research and Practice (2011)
4. Eisenbarth, T., Koschke, R., Simon, D.: Locating features in source code. IEEE Trans. Softw. Eng. 29(3), 210–224 (Mar 2003)
5. Eisenberg, A., De Volder, K.: Dynamic feature traces: finding features in unfamiliar code. In: Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on. pp. 337–346 (Sept 2005)
6. Font, J., Arcega, L., Haugen, Ø., Cetina, C.: Building software product lines from conceptualized model patterns. In: Proceedings of the 2015 19th International Software Product Line Conference. SPLC '15, Nashville, TN, USA. (2015)
7. Haugen, Ø., Mller-Pedersen, B., Oldevik, J., Olsen, G.K., Svendsen, A.: Adding standardized variability to domain specific languages. In: Proceedings of the 2008 12th International Software Product Line Conference. pp. 139–148. SPLC '08, IEEE Computer Society, Washington, DC, USA (2008)
8. IBM: An architectural blueprint for autonomic computing. Tech. rep., IBM (2006)
9. Martinez, J., Ziadi, T., Bissyand, T.F., Le Traon, Y.: Bottom-up adoption of software product lines a generic and extensible approach. In: Proceedings of the 2015 19th International Software Product Line Conference. SPLC '15, Nashville, TN, USA. (2015)
10. Muoz, J.: Model Driven Development of Pervasive Systems. Building a Software Factory. Ph.D. thesis, Universidad Politcnica de Valencia (2008)
11. Revelle, M., Dit, B., Poshyvanyk, D.: Using data fusion and web mining to support feature location in software. In: Program Comprehension (ICPC), 2010 IEEE 18th International Conference on. pp. 14–23 (June 2010)
12. Wilde, N., Scully, M.C.: Software reconnaissance: Mapping program features to code. Journal of Software Maintenance 7(1), 49–62 (Jan 1995)
13. Wille, D., Holthusen, S., Schulze, S., Schaefer, I.: Interface variability in family model mining. In: Proceedings of the 17th International Software Product Line Conference Co-located Workshops. pp. 44–51. SPLC '13 Workshops, ACM, New York, NY, USA (2013)
14. Zhang, X., Haugen, O., Moller-Pedersen, B.: Model comparison to synthesize a model-driven software product line. In: Software Product Line Conference (SPLC), 2011 15th International. pp. 90–99 (Aug 2011)

12.2 MRT'16 Paper

Title: An Infrastructure for Generating Run-time Model Traces for Maintenance Tasks.

Authors: Lorena Arcega, Jaime Font, Øystein Haugen, Carlos Cetina.

Proceedings: Proceedings of the 11th International Workshop on Models@run.time.

Location: Saint Malo, France - October 4, 2016

Publisher: CEUR Workshop Proceedings

Pages: 35-42

DOI: `http://ceur-ws.org/Vol-174`

Contribution: Lorena Arcega is the main author of the paper and is responsible for 90% of the work. She was also responsible for the oral presentation of the work which took place during the conference.

An Infrastructure for Generating Run-time Model Traces for Maintenance Tasks

Lorena Arcega*†, Jaime Font*†, Øystein Haugen‡ and Carlos Cetina*
*Universidad San Jorge, SVIT Research Group, Zaragoza, Spain
Email: {larcega,jfont,ccetina}@usj.es
†University of Oslo, Department of Informatics, Oslo, Norway
‡Østfold University College, Department of Information Technology, Halden, Norway
Email: oystein.haugen@hiof.no

***Abstract*—Current research efforts are focused on taking advantage of the models at run-time for run-time decision-making related to run-time system concerns associated with autonomic and adaptive systems. In addition, all systems need maintenance over time as new requirements emerge or when bug-fixing becomes necessary. Models at run-time can provide an execution trace of a high level of abstraction that is useful for maintenance tasks. In this paper, we propose a generic infrastructure, which is able to get the run-time model trace. Our infrastructure creates a descriptive model of the running code by means of Code-Model Connection Rules. These rules translate the behaviour of the running source code in model traces. We validate our infrastructure in a Smart Hotel. The results of our infrastructure show promising results towards the generation of model traces from source code at run-time. However, further work is required to a better specification of the rules, to solve some issues with the model dependencies and to allow the propagation of changes in the models to the source code.**

I. Introduction

Models at run-time research efforts are concerned with how abstractions of software implementations can be used at run-time to manage software changes at run-time [1]. Current research works are focused on how models can be used to support run-time adaptations in autonomous systems (i.e., in self-* systems) [2]. Usually, the adaptation actions are driven by a MAPE-K loop [3] and prescriptive models. That is, changes in the model are transferred to the system.

In software development, all systems evolve over time as new requirements emerge or when bug-fixing becomes necessary. Lehman et al. [4] pointed out that up to the 80% of the lifetime of a system is spent on maintenance and evolution activities. Currently, research efforts in feature location are concerned with identifying software artifacts associated with a program functionality (a feature). Feature location is one of the most important and common activities performed by developers during software maintenance and evolution [5].

The amount of run-time data produced by many applications tends to be huge and is recorded in different forms. Relevant information has to be extracted and stored with the corresponding artifacts produced at design-time to reveal the causes of unexpected software behavior [6]. The purpose of the analysis is to find the effects within a chain of effects occurred during software execution [7]. Therefore, some techniques are needed to extract and understand the run-time information. Then, run-time information is necessary at design-time to make decision regarding the system maintenance.

The goal of our work is to extend the use of models at run-time to perform maintenance task. This work presents our generic infrastructure to generate run-time model traces through the use of connection rules specified at design-time called *Code-Model Connection Rules*.

The *Code-Model Connection Rules* maps model elements to different point in the source code such as methods or values of variables. Our generic infrastructure is divided in two parts: Controller and Translator. It controls the implementation code, check it through the *Code-Model Connection Rules*, and, if changes in the run-time model are necessary, our generator translates the changes in the model at run-time.

We validate our infrastructure in our Smart Hotel. The infrastructure creates model traces from source-code to models. The results of the validation of our infrastructure show promising results towards the generation of model traces from source code at run-time. However, further work is required to a better specification of the *Code-Model Connection Rules*, to solve some issues with the model dependencies and to allow the performance of maintenance tasks at model level, that is, the propagation of changes in the models to the source code.

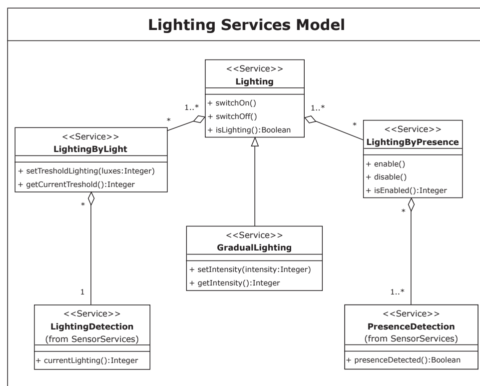
The remainder of the paper is structured as follows. In Section II, we introduce the Smart Hotel. In Section III, we describe the steps of our approach. In Section IV, we validate our approach with the Smart Hotel case study and then discuss the results. In Section VI, we examine the works related with this one. Finally, in Section VII, the conclusions of our work are presented.

II. Background

The running example and the evaluation of this paper are performed through a Smart Hotel [8]. In this section, we present the models that are used by the Smart Hotel.

We use Pervasive Modeling Language (PervML) [9] to describe the Smart Hotel. PervML[1] is a Domain Specific Language (DSL) that describes pervasive systems using high-level abstraction concepts such as services. This language

[1] https://tatami.dsic.upv.es/pervml/index.php



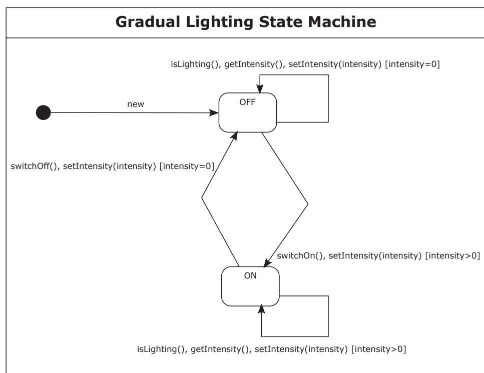


Fig. 1. Smart Hotel PervML Models

is focused on specifying heterogeneous services in distinct physical environments. This DSL has been applied to develop solutions in the Smart Hotel domain.

PervML uses six different models which globally provide the information that is required to describe a pervasive system. These models are:

- The **Services Model** describes the kinds of services that are provided in the pervasive system. It is used for specifying the common characteristics of these similar services.
- The **Structural Model** indicates the number and kind of services that are provided by the pervasive system in every system location.
- The **Interaction Model** is used for describing the communication that is produced between the different components of the services.
- The **Binding Providers Model** is used to describe the different kinds of devices or software systems that are in charge of providing the basic pervasive system functionality.
- The **Components Structural Specification** is used to assign devices and software systems to each one of the service components.
- The **Components Functional Specification** is used to specify the actions that are executed when an operation of a service is invoked.

Because of space constraints, Fig. 1 shows only a small part of the PervML models related to the functioning of the lights in the Smart Hotel. The concrete syntax of PervML is similar to the one used by UML. The Smart Hotel provides services which offer operations for switching on and switching off the lights or setting the light intensity. The Gradual Lighting State Machine shows the state machine related to the gradual lights. This kind of service supports setting the light intensity from 0% to 100% by means of a set of operations.

The output system is obtained through a model to text transformation. That is, each element of the model is translated to one or more Java bundles. In addition, we use an implementation framework of PervML and the OSGI framework [10]. See [9] for more implementation details.

III. From Source-Code to Models

This work presents our generic infrastructure to generate run-time model traces through the use of connection rules specified at design-time called *Code-Model Connection Rules*. The infrastructure monitors the software changes in the running system and put the running information into a descriptive model at run-time.

Fig. 2 shows an overview of our Run-time Model Trace Generator. At design-time, the software engineer creates the models that specify the system and are used to generate the run-time model. From the models, the code is implemented, usually manually by using the models as a reference and in some cases it is doing automatically or semi-automatically. The software engineer also species the rules, *Code-Model Connection Rules*, which will generate the descriptive model at run-time.

In a regular use, the deployment of the system is performed running the source code. When the software engineer wants to obtain the run-time model trace, he can activate the Run-time Model Trace Generator.

In the Run-time Model Trace Generator, our infrastructure controls the implementation code, checks it through the *Code-Model Connection Rules*, and, if changes in the run-time model are necessary, our generator translates the changes in the model trace at run-time.

As a summary, the key idea of our infrastructure is a loop which is responsible for translating the information of the run-time knowledge from source code to models. Next subsections present the *Code-Model Connection Rules* used in our infrastructure and the steps of our Run-time Model Trace Generator.

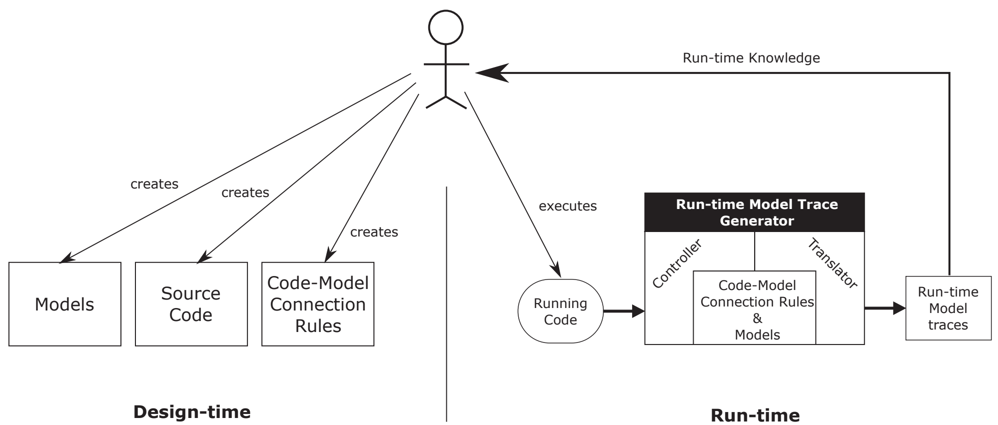


Fig. 2. Overview of the Run-time Model Trace Generator

A. The Code-Model Connection Rules

To generate a model from the executed source code, we developed a *Code-Model Connection Rules*, which are expressed using Event-Condition-Action (ECA) rules [11]. We are not reinventing the ECA rules. The ECA rules have been intensely studied and used for the management of self-adaptive systems [8]. However, we use these ECA rules in a novel way to perform changes in a run-time model when the code is executed.

The definition of our rules is based on the ERA language [12] to provide mechanism for specifying the execution of external actions (actions in the run-time model) instead of internal actions (actions in the running code of the system).

The *Code-Model Connection Rules* take the following shape: *'when a line of the source code is executed, the state of variables change, or both take place, then a model element should be created, modified, or deleted in the run-time model'*.

For example, when the line of code that creates the gradual lighting service *(GradualLightService mainGradualLightService=new GradualLightService())* is executed, the GradualLighting model element in the run-time model must appear. In this particular case, the rule is depicted as follows:

```
(GradualLightService mainGradualLightService=new
    GradualLightService(),
    IS_NOT(LightingServices.GradualLighting),
    ADD_ELEMENT(LightingServices.GradualLighting));
```

In this rule, the first parameter (*GradualLightService mainGradualLightService=new GradualLightService()*) is the line of code that triggers the rule. The second parameter (*IS_NOT(LightingServices.GradualLighting)*) indicates the condition that must be fulfilled and in which model. The keyword *IS_NOT* checks if the model element appears in the run-time model and return true if it is not in the model. In this case it returns true if the model element GradualLighting is not in the LightingServices model. The third parameter (*ADD_ELEMENT(Lighting- Services.GradualLighting)*) is the kind of modification that must be performed in the run-time model. In the smart hotel, it is the addition of the model element GradualLighting in the LightingServices model.

On the other hand, when the code indicates that the GradualLightingService is not available any longer (*remove(mainGradualLightService)*), the element must be removed from the run-time model. The rule is similar to the one for adding model elements:

```
(remove(mainGradualLightService),
    IS( LightingServices .GradualLighting ),
    DEL_ELEMENT(LightingServices.GradualLighting));
```

The first parameter (*remove(mainGradualLightService)*) is the line of code that must be executed to trigger the Model-Code Connection rule. The second parameter *(IS(LightingServices.GradualLighting))* indicates if the model element is in the model. The third parameter (*DEL_ELEMENT(LightingServices.Gradual- Lighting)*) is the modification that must be performed in the run-time model. In this case it is a deletion of the model element GradualLighting in the LightingServices model.

Although the idea of the *Code-Model Connection Rules* is domain independent, our current implementation is tailored to the Smart Hotel. Even though these rules have helped us to address the research questions, our future work involves designing independent domain rules that combine debug expressiveness to specify the state of the variables and execution points (triggers) and OCL-like expressions to specify (conditions and actions in the run-time model).

Some model elements may require few rules to be synchronized with the code, but others may require more effort.

These rules are a trade-off, the more specifications the engineer makes, the richer the run-time model will be, but at the cost of the increase in the amount of time that must be spent by the software engineer in order to specify them.

One of the main characteristics of our approach is that, by means of the *Code-Model Connection Rules*, the software engineer can change the set of the models defined at design-time that will be used at run-time. That is, the software engineer can decide how many model elements should be connected with the running code.

B. Run-time Model Trace Generator

Szvetis and Zdum [13] creates a classification of the objectives, techniques, kinds and architectures used with models at run-time. According to their categories, we can classify our infrastructure to work with models at run-time as follows:

- Objectives: The main one is monitoring the system by using models in order to help the understanding of the system behavior. In addition, we use models to raise the abstraction level and bring closer the problem to the user. Finally, the use of models provides platform-independent views of the system under observation.
- Technique: We use the introspection technique that deals with extraction of run-time system data in order to apply model-based techniques on the analyzed behavior. In addition, they subdivided the introspection in three main groups: event log checking, instrumentation and management API.
  Specifically, our infrastructure is similar to the instrumentation group. Even though, we do not insert any extra source code to the system. We use the rules to translate the running software of a system into a model.
- Kind: Our infrastructure does not depend on one kind of model. We use a domain specific language (PervML, see section II), however, the model generated depends on the defined rules and the user can choose whoever he wants.
- Architecture: We use a model-aware middle-ware architecture. This architecture allows us to monitor the system while our solution checks the rules and creates the model at run-time.

The Code-Model Connection rules previously presented enables to reflect running-code changes into a descriptive run-time model. The details of our infrastructure are depicted in Fig. 3.

- **Controller:** When the system is running, the running source code needs to be controlled (see Fig. 3). The implementation of the controller is developed by means of the Java Code Language Introspection Capabilities [14].
  Our infrastructure uses the run-time state of the code as input to check conditions from the *Code-Model Connection Rules*. In the running example, our approach detects every time that the line *'GradualLightService mainGradualLightService = new GradualLightService()'* is executed.
  Once a relevant change in the state of the code is detected, our approach checks if the conditions of the *Code-Model Connection Rule* is fulfilled. In the running example, our approach checks if the GradualLighting model element appears in the LightingServices run-time model, (*'IS_NOT(LightingServices.GradualLighting)'*).
- **Translator:** The response of the controller is used to translate the source code into models (see Fig. 3). The implementation of the translator is developed by means of the EMF Introspection Capabilities [15].
  Our infrastructure uses the action of the *Code-Model Connection Rule* to determine the modifications that must be performed in the run-time model. In the running example, the action determines that the GradualLighting model element must be added to the LightingServices run-time model (*'ADD_ELEMENT(LightingServices.GradualLighting)'*). If the model element does not appear in the run-time model, it checks the design-time models and takes the information that is needed in order to add this model element to the run-time model. If the model element appears in the run-time model, our approach checks if some modifications are necessary.
  The run-time model is modified in order to add, modify or delete the model elements specified in the *Code-Model Connection Rules*. In the running example, the GradualLighting model element is added to the run-time model.

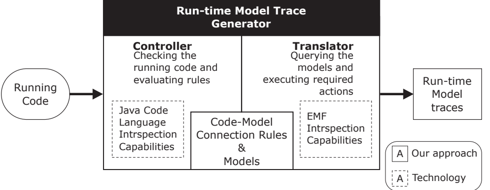


Fig. 3. Details of the Run-time Model Trace Generator

The model generated at run-time only includes elements that have been executed. This run-time model is the main artifact that the software engineer can use to improve his knowledge of the system. We extract the run-time representation of the system as models to enable reasoning on a high level of abstraction.

IV. Validation

We validate whether our infrastructure creates the correct models at run-time. To do that, we compare our model traces generated at run-time with an oracle that is composed by the models of the defined scenarios in [8]. We generate a run-time model trace while the other model trace is extracted from a previous work [8], we compare both traces in order to detect differences.

We defined the experimental design of our study using the Goal-Question-Metric method (GQM) [16]. We used the template presented in [17]. The GQM method was defined as a mechanism for defining and interpreting a set of operation

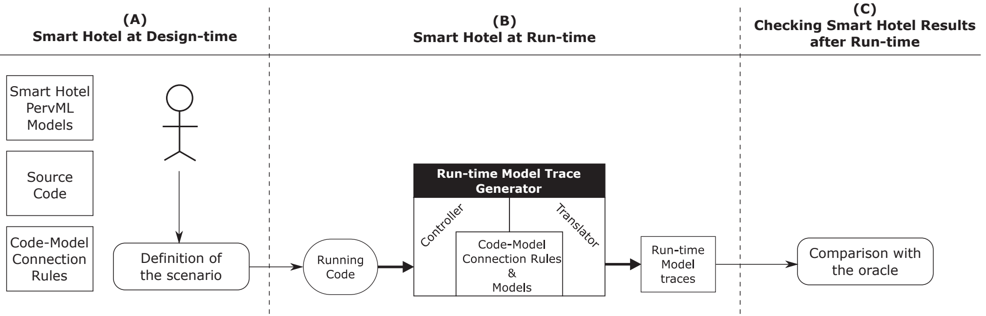


Fig. 4. The validation process followed in the Smart Hotel

goals using measurements. In this evaluation, our goal was the following:

- Object: Our Smart Hotel
- Purpose: Validation
- Issue: The correctness of the models generated at run-time
- Context: The infrastructure for of our Run-time Model Trace Generator

To fulfill this goal, we focused on answering the following research question:

1) Does our infrastructure generate the same model traces as the oracle?

Basili in [16] and Travassos in [18] describe four kinds of studies: in-vivo, in-vitro, in-virtuo, and in-silico. In our case, we chose to carry out in-virtuo experiments, where the real world is described as computer models. This experiment involves the interaction among participants and a computerized model of reality. The simulated environment offers major advantages regarding the cost and the feasibility of replicating a real-world configuration. In addition, some scenarios such as res or floods cannot be replicated in the real world.

The Smart Hotel is developed by means of a software product line, the PervML model and the source code of the Smart Hotel are configured with a feature model. The feature model specifies the 39 different features that the Smart Hotel has implemented. We use the PervML model traces of the [8] approach as an oracle to evaluate the presented approach. That is, we make use of a set of implementation codes whose model realizations are known beforehand and will be considered as the ground truth, enabling us to compare the run-time traces extracted from the [8] approach with the results of the models generated by our infrastructure.

Figure 4 shows the entire process that we have followed in this validation. For the validation, we used the Smart Hotel defined in [8]. The Smart Hotel presents 476 model elements in the design-time models. It consists of 268 classes implemented in about 67,207 methods of Java source code. In the evaluation set-up, a simulated environment was used to represent the Smart Hotel. In addition, at design-time, the *Code-Model Connection Rules* and the scenario are defined to allow the generation of the trace model from the code at run-time, as shown in Fig. 4 (A).

After running the scenario, see Fig. 4 (B), our Run-time Model Trace Generator (see Fig. 4 left) generates the run-time model. The infrastructure monitors the running code and applies the corresponding *Code-Model Connection Rules* to translate source code behavior into a descriptive run-time model trace.

For the comparison, see Fig. 4 (C), we collect the traces by means of the Reconfiguration Tracker that is provided by the implementation of the Smart Hotel that we used [8].

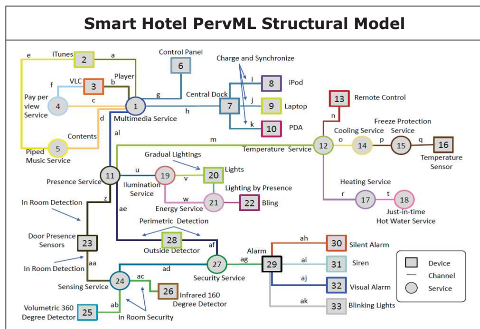
At the end, we have two set of models that represents the behavior of the system. One of the set is generated by our Run-time Model Generation and the second set is the oracle. Then, we can compare the results of the two approaches.

The number of model elements in the run-time model depends on the *Code-Model Connection Rules* defined. As we wanted to determine if the models generated was equivalent, we performed the validation with a set of rules that connected all the possible elements with the source code. By this way, we could compare both models traces easily.

Therefore, the number of rules that the software engineer must use is an open topic that will be addressed in the future. The specification of the rules takes times and it is not always necessary to specify all of them. Thus, I hope I can guide the software engineer about what are the rules he need to specify.

A concrete example of our validation is shown in Fig. 5. The Smart Hotel is defined with a large number of models (see section II), however in this section we only show one of these models. As the examples show, all the models obtained are a subset of the entire definition of the Smart Hotel.

The fist one of the models of Fig. 5, Smart Hotel PervML Structural Model, corresponds to the one that defines the entire system of the Smart Hotel. The second one, Smart Hotel PervML Structural Model in Watching a Movie Scenario represents the model generated in the Watching a Movie



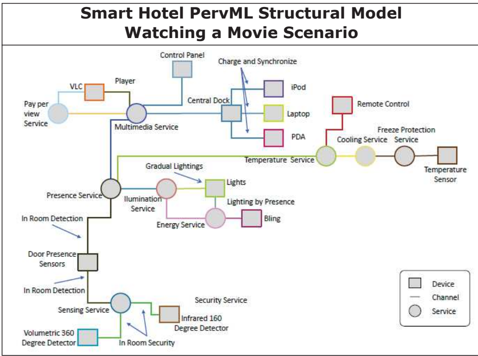


Fig. 5. Smart Hotel PervML Structural Models

Scenario.

The definition of the Watching a Movie scenario is as follows:

'The user of the hotel decides to watch a movie from the of which the hotel offers. The user can select the movie in the screen of the control panel. In addition, the user will be able to choose the room preferences (illumination and temperature) for a better experience. The system will give him many options so he can choose the one that suits him more. The user is enjoying the movie but, someone is calling to his cell phone with Bluetooth technology synchronized with the rooms main system. This call will be notified through the main screen. He is forced to stop watching the movie but he will have the option to continue where he left at any other time during his stay.'

Then, the main devices that are involved in this scenario are the cell phone, the control panel, the central dock. Some other devices related to this scenario are ones related to the illumination and temperature services. In the same way, the main service in this scenario is the multimedia service, and also the ones related to illumination and temperature.

This trace generated from this scenario is formed by 23 channels, 13 devices and 9 services in the Smart Hotel PervML Structural model (see bottom part of Fig. 5).

A. Does our infrastructure generate the same model traces as the oracle?

The results of our infrastructure show promising results towards the generation of model traces from source code at run-time.

The results obtained in this validation suggest that our infrastructure generates a valid model trace from the running scenarios. All the generated traces contain the main devices and services necessary for execute the defined scenario.

However, the trace generated with our infrastructure contained more information than the trace from the oracle. But, the trace from the oracle was contained in the trace generated with our infrastructure. The major difference between our generated model traces at run-time and the traces from the oracle lies in the number of attributes of the model element.

In addition, it is necessary further specific experiments in order to identify more relevant correlations between the source code and the run-time models. However, the *Code-Model Connection Rules* have flexibility to enable and disable the model elements that must appear in the run-time model trace.

Another issue detected is related to the dependencies between model elements. The management of the dependencies through the use of the *Code-Model Connection Rules* is cumbersome and error prone. We detect that a dependency tree could be very useful to check the consistency of the model trace while it is being generated.

Moreover, our infrastructure generates the model trace to increase the level of abstraction and to ease the maintenance task. But nevertheless, the maintenance task must be performed at source code level because the infrastructure does not allow propagate changes in the models to the source code of the system.

In spite of the results, this initial effort must not be underestimated since the Run-time Model Trace Generator produced proper output to see the first results. The overall performance of the infrastructure needs more tests. Further work is required to see if these concerns are reasonable and if so, find out how to these previous problems can be solved.

V. THREATS TO VALIDITY

We use the classification of threats to validity in [19]. This classification distinguishes four aspects of validity:

1) **Construct validity:** This aspect of validity reflects the extent to which the operational measures that are studied represent what the researchers have in mind and what is investigated based on the research questions.
   The purpose validation does not have a true/false answer. To minimize this threat the scenarios were designed by two experts in the Smart Hotel. These experts have developed the Smart Hotel and in other model-based tools (in the induction hob domain and train control software domain). Their participation was limited to the design of the scenarios.

The measure used in our research is the similarity of our model traces extracted and the oracle. These similarity measures have been proved in other kind of models with fair results [20].

2) **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.
   In our work, this is a very low risk due to the human task are reduced to the minimum. The comparison between the traces were performed automatically by a computer, therefore the humans did not affect the final result.
3) **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.
   The software system used in the validation is representative of those used in practice. Given the scale and complexity of our Smart Hotel, we consider our evaluation a good starting point representing a realistic case. However, this threat can be reduced if we experiment with other software systems of different sizes and domains.
4) **Reliability:** This aspect is concerned with to what extent the data and the analysis are dependent on the specific researchers.
   The main risk relies on the selection of the running scenarios to obtain the execution trace of the models. Since we are experts in the Smart Hotel system, we can claim that our scenarios are good representatives of the entire system, and, therefore, our infrastructure generates the correct models for all the system. However, depending on the chosen scenarios, the result may differ.

VI. Related Work

Some approaches use the run-time models to visualize and analyze data from model-based systems. Although they do not generate the run-time models, their works are based on extracting information from run-time models.

Graf and Mller-Glaser [21] define various debugging-perspectives independent of the actual execution platform for identifying error occurrences. They extract run-time information out of the executed binary from the target platform and transport this information back to the model level. Obtained run-time information can then be used to visualize the internal state of the executable. In contrast to our approach, they extend the UML meta-model with meta-classes that allow storage of data acquired by the model mapping.

Maoz [22] introduces an approach which uses model-based traces to record the run-time of a system through abstractions provided by models used in its design. The traces include information about the systems from which they were generated, the models that induced them and the relationships between them at run-time. The approach is focused on metrics and operators for such traces to understand the structure and behavior of a running system while we are focused on the infrastructure to generate the run-time model from source code.

Bodenstaff et al [23] uses event log checking to check a model against the running system. They assume that the event log is consistent with the running system and also with a model if essential parts of the model do not contradict. In contrast to our approach, they are focus on identifying relevant parts of the log event and extract them as models while our approach generates the models directly from the running system.

Szvetits et al. [6] propose the usage of run-time models in combination with model-driven techniques and interactive visualizations to ease tracing between log file entries and corresponding software artifacts. They creates a repository-based approach to augment root cause analysis with interactive tracing views to maximize the re-usability of models created during the software development process. In contrast to our approach, they focus their work on the interactive visualizations while our work is focused on the generation of run-time models from non-model-based system. However, we can apply their techniques to analyze the model traces that we extract from the running system.

Some other approaches insert monitoring functionality into the system in order to observe the system behavior.

Nierstrasz et al. [24] use instrumentation at run-time in combination with and Abstract Syntax Tree (AST) model as abstraction above the byte-code level. They use the concept of links to be set as annotations to AST nodes which are transferred by a compiler plugin before execution. This results in code to be inserted in the program at nodes where links are installed. Such links can also be installed by other links or manipulated by themselves.

Hamann et al. [25] propose monitoring of JVM hosted applications by using platform-aligned model (PAM) as link between the platform-specific model (PSM) and the platform-independent model (PIM). A PAM is iteratively designed through assumptions which are stated, checked, and refined.

This kind of approaches requires the source code of the system to be available. In contrast to us, this approaches extract the run-time data and then they translate the extracted data as models to enable reasoning on a high level of abstraction while we extract the data as models at run-time and do not introduce anything new in the source code.

VII. Conclusion

This work proposes a generic infrastructure to generate run-time models through the use of connection rules called *Code-Model Connection Rules*. Specifically, our infrastructure allows change from source-code to descriptive models increasing the abstraction level.

Our validation in our Smart Hotel compares the result obtained from our infrastructure and from a oracle. We use the model traces generated to determine if the results obtained are correct. The results of the validation of our infrastructure show promising results towards the generation of model traces from source code at run-time. However, further work is required to a better specification of the *Code-Model Connection Rules*, to

solve some issues with the model dependencies and to allow the performance of maintenance tasks at model level, that is, the propagation of changes in the models to the source code.

In addition to the previous one, our future work involves designing independent domain rules that combine debug expressiveness to specify the state of the variables and execution points (triggers) and OCLlike expressions to specify (conditions and actions in the run-time model).

Acknowledgment

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

References

[1] U. Aßmann, N. Bencomo, B. H. C. Cheng, and R. B. France, "Models@run.time (Dagstuhl Seminar 11481)," *Dagstuhl Reports*, vol. 1, no. 11, pp. 91–123, 2012. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2012/3379

[2] H. Giese, N. Bencomo, L. Pasquale, A. J. Ramirez, P. Inverardi, S. Wätzoldt, and S. Clarke, *Models@run.time: Foundations, Applications, and Roadmaps*. Cham: Springer International Publishing, 2014, ch. Living with Uncertainty in the Age of Runtime Models, pp. 47–100.

[3] IBM, "An architectural blueprint for autonomic computing," IBM, Tech. Rep., 2006.

[4] M. M. Lehman, J. Ramil, and G. Kahen, "A paradigm for the behavioural modelling of software processes using system dynamics," Imperial College of Science, Technology and Medicine, Department of Computing, Tech. Rep., Sep 2001.

[5] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: A taxonomy and survey," in *Journal of Software Maintenance and Evolution: Research and Practice*, 2011.

[6] M. Szvetits and U. Zdun, "Enhancing root cause analysis with runtime models and interactive visualizations," in *8th International Workshop on Models at Run.time*, Miami, Florida, USA, September 2013. [Online]. Available: http://eprints.cs.univie.ac.at/3764/

[7] H. W. Dettmer, *Goldratt's theory of constraints: a systems approach to continuous improvement*. ASQ Quality Press, 1997.

[8] C. Cetina, "Achieving autonomic computing through the use of variability models at run-time," Ph.D. dissertation, Universidad Politécnica de Valencia, 2010.

[9] J. Muñoz, "Model driven development of pervasive systems. building a software factory," Ph.D. dissertation, Universidad Politécnica de Valencia, 2008.

[10] D. Marples and P. Kriens, "The open services gateway initiative: an introductory overview," *IEEE Communications Magazine*, vol. 39, no. 12, pp. 110–114, Dec 2001.

[11] J. O. Kephart and W. E. Walsh, "An artificial intelligence perspective on autonomic computing policies," in *Proceedings of the 5th IEEE International Workshop on Policies for Distributed Systems and Networks. POLICY '04*, June 2004, pp. 3–12.

[12] J. J. Alferes, F. Banti, and A. Brogi, *Logics in Artificial Intelligence: 10th European Conference, JELIA 2006 Liverpool, UK, September 13-15, 2006 Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, ch. An Event-Condition-Action Logic Programming Language, pp. 29–42.

[13] M. Szvetits and U. Zdun, "Systematic literature review of the objectives, techniques, kinds, and architectures of models at runtime," *Software & Systems Modeling*, vol. 15, no. 1, pp. 31–69, 2016. [Online]. Available: http://dx.doi.org/10.1007/s10270-013-0394-9

[14] K. Arnold, J. Gosling, and D. Holmes, *The Java programming language 4th Edition*. Addison-wesley, 2016.

[15] "The eclipse project: Emf model query." [Online]. Available: https://projects.eclipse.org/projects/modeling.emf.query

[16] V. R. Basili, "The role of experimentation in software engineering: Past, current, and future," in *Proceedings of the 18th International Conference on Software Engineering*, ser. ICSE '96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 442–449. [Online]. Available: http://dl.acm.org/citation.cfm?id=227726.227818

[17] V. R. Basili, G. Caldiera, and H. D. Rombach, "The goal question metric approach," in *Encyclopedia of Software Engineering*. Wiley, 1994.

[18] M. O. B. G. H. Travassos, "Contributions of in virtuo and in silico experiments for the future of empirical studies in software engineering," in *In Proceedings of the Workshop on Empirical Studies in Software Engineering (ESEIW)*. IEEE Computer Society, 2003.

[19] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Softw. Engg.*, vol. 14, no. 2, pp. 131–164, Apr. 2009. [Online]. Available: http://dx.doi.org/10.1007/s10664-008-9102-8

[20] M. Ehrig, A. Koschmider, and A. Oberweis, "Measuring similarity between semantic business process models," in *Proceedings of the Fourth Asia-Pacific Conference on Comceptual Modelling - Volume 67*, ser. APCCM '07. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2007, pp. 71–80. [Online]. Available: http://dl.acm.org/citation.cfm?id=1274453.1274465

[21] P. Graf and K. D. Muller-Glaser, "Dynamic mapping of runtime information models for debugging embedded software," in *Seventeenth IEEE International Workshop on Rapid System Prototyping (RSP'06)*, June 2006, pp. 3–9.

[22] S. Maoz, "Using model-based traces as runtime models," *Computer*, vol. 42, no. 10, pp. 28–36, Oct 2009.

[23] L. Bodenstaff, A. Wombacher, M. Reichert, and R. Wieringa, "Made4ic: an abstract method for managing model dependencies in inter-organizational cooperations," *Service Oriented Computing and Applications*, vol. 4, no. 3, pp. 203–228, 2010. [Online]. Available: http://dx.doi.org/10.1007/s11761-010-0062-7

[24] O. Nierstrasz, M. Denker, and L. Renggli, *Model-Centric, Context-Aware Software Adaptation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 128–145. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-02161-9_7

[25] L. Hamann, M. Gogolla, and M. Kuhlmann, "Ocl-based runtime monitoring of jvm hosted applications," *Electronic Communications of the EASST*, vol. 44, 2011.

12.3 SAM'16 Paper

Title: Feature Location Through the Combination of Run-Time Architecture Models and Information Retrieval.

Authors: Lorena Arcega, Jaime Font, Øystein Haugen, Carlos Cetina.

Proceedings: Proceedings of the 9th International Conference on System Analysis and Modeling. Technology-Specific Aspects of Models.

Location: Saint Malo, France - October 3-4, 2016

Publisher: CEUR Workshop Proceedings

Pages: 180-195

DOI: `https://doi.org/10.1007/978-3-319-46613-2_12`

Contribution: Lorena Arcega is the main author of the paper and is responsible for 90% of the work. She was also responsible for the oral presentation of the work which took place during the conference.

Feature Location Through the Combination of Run-Time Architecture Models and Information Retrieval

Lorena Arcega[1,2(✉)], Jaime Font[1,2], Øystein Haugen[3], and Carlos Cetina[1]

[1] SVIT Research Group, Universidad San Jorge, Zaragoza, Spain
{larcega,jfont,ccetina}@usj.es
[2] Department of Informatics, University of Oslo, Oslo, Norway
[3] Department of Information Technology, Østfold University College, Halden, Norway
oystein.haugen@hiof.no

Abstract. Eighty percent of the lifetime of a system is spent on maintenance activities. Feature location is one of the most important and common activities performed by developers during software maintenance. This work presents our approach for performing feature location by leveraging the use of architecture models at run-time. Specifically, the execution information is collected in the architecture model at run-time. Then, our approach performs an Information Retrieval technique at the model level. We have evaluated our approach in a Smart Hotel with its architecture model at run-time. We compared our architecture-model-based approach with a source-code-based approach. The rankings produced by the approaches indicate that since models are on a higher abstraction level than source code, they provide more accurate results. Our architecture-model-based approach ranks the relevant elements in the top ten positions of the ranking in 84 % of the cases; in the top positions in the ranking of the source-code-based approach, there are false positives associated with some programming patterns and true positives are spread between positions 12 and 100.

Keywords: Arquitecture model · Models@Run-time · Feature location · Information retrieval · Reverse engineering

1 Introduction

In software development, all systems evolve over time as new requirements emerge or when bug-fixing becomes necessary. Lehman et al. [13] pointed out that up to 80 % of the lifetime of a system is spent on maintenance and evolution activities. Feature location is one of the most important and common activities performed by developers during software maintenance and evolution [8]. Currently, research efforts in feature location are concerned with identifying software artifacts that are associated with a program functionality (a feature).

© Springer International Publishing AG 2016
J. Grabowski and S. Herbold (Eds.): SAM 2016, LNCS 9959, pp. 180–195, 2016.
DOI: 10.1007/978-3-319-46613-2_12

Models at run-time provide a kind of formal basis for reasoning about the current system state, for reasoning about necessary adaptations, and for analyzing the consequences of possible system adaptations. Models at run-time development approaches have the proven capability to deliver complex, dependable software effectively and efficiently. In this paper, we show that the information extracted from architecture models at run-time can be useful in the field of feature location. In models at run-time [5], there is a causal connection between the system and the run-time model (i.e., there is a bidirectional relation between the source code and the run-time model).

This work proposes an approach that combines architecture models at run-time and Information Retrieval (IR) for feature location. In the first step of our approach, the software engineer executes a scenario, which uses the desired feature to be located. The execution information is collected in the architecture model at run-time. Then, our approach filters the trace in order to extract the relevant elements of the models. We adapt an information retrieval technique, Latent Semantic Indexing (LSI). This technique allows the software engineers to write queries that are relevant to the feature to be located. Finally, the software engineers obtain a ranked list of model elements that are related to the feature based on the similarity to the query.

We have evaluated our approach in a Smart Hotel that is defined with an architecture model at run-time. The Smart Hotel presents sixty-eight model elements in the architecture model that are implemented in 268 Java classes (about 67,207 methods of source code). We have compared our approach based on models with a feature location approach that is based on source code, which is presented in [14]. We chose this approach because it outperforms all other source-code-based approaches that use a single scenario and information retrieval [8].

The results indicate that the information gathered at a high level of abstraction of architecture models is closer to natural language queries of software engineers; hence, the rankings are more accurate. Our architecture-model-based approach ranks the relevant elements in the top ten positions of the ranking in 84 % of the cases; in the top positions in the ranking of the source-code-based approach, there are false positives associated with some programming patterns and true positives are spread between positions 12 and 100.

The remainder of the paper is structured as follows. In Sect. 2, we present the Smart Hotel. In Sect. 3, we introduce our approach for feature location with architecture models at run-time. In Sect. 4, we evaluate our approach with the Smart Hotel and we discuss the results. In Sect. 5, we examine the related work of the area, and we present our conclusions in Sect. 6.

2 The Smart Hotel

The running example and the evaluation of this paper are performed through a Smart Hotel [6]. The Smart Hotel is reconfigured in response to changes in the context, for example if there is a client in the room or not, and what activities they may be performing (sleeping, watching TV, ...). This section shows the

language used for specifying the architecture model of the Smart Hotel. This section also shows how the architecture model is reconfigured at run-time in response to context changes.

2.1 The Architecture Model

We use Pervasive Modeling Language (PervML) [16] to describe the Smart Hotel architecture. PervML[1] is a DSL that describes pervasive systems using high-level abstraction concepts based on Meta-Object Facility (MOF)[2]. This language is focused on specifying heterogeneous services in specific physical environments such as the services of a Smart Hotel. This DSL has been applied to develop solutions in the Smart Hotel domain. The PervML language provides different models to specify the services and devices of a pervasive system.

Due to space constraints, in this paper, we only focus on the subset of PervML that specifies the relationships among devices and services. This subset specifies the components that define a particular configuration system (services and devices) and how these components are connected with each other (channels). Services are depicted by circles, devices are depicted by squares, and the channels connecting services and devices are depicted by lines (see Fig. 1).

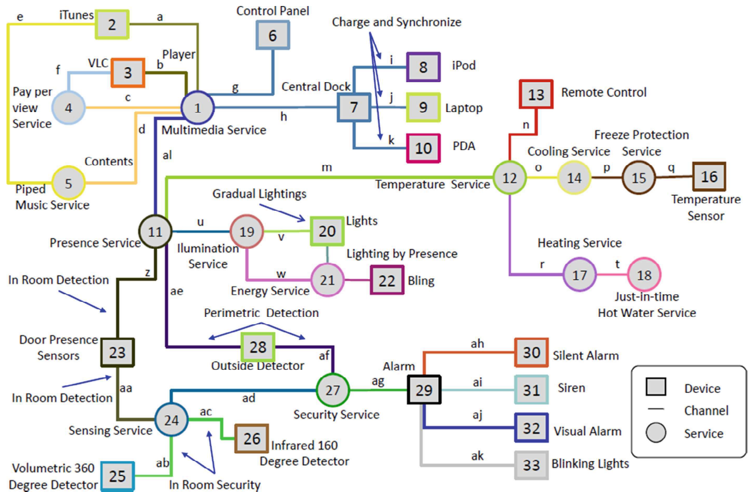


Fig. 1. Smart hotel architecture model

[1] https://tatami.dsic.upv.es/pervml/index.php.
[2] Meta object facility (MOF) 2.0 core specification, 2003.

2.2 The Architecture Model Reconfiguration

The Smart Hotel reconfiguration engine determines how the system should be reconfigured in response to a context change, and then it modifies the PervML architecture model accordingly. The Monitor uses the run-time state as input to check context conditions. If any of these conditions are fulfilled, the Analyzer queries the run-time models about the necessary modifications. The response of the models is used by the Planner to elaborate a reconfiguration plan. This plan also contains reconfiguration actions, which modify the architecture model and maintain the consistency between the PervML architecture model and the system. The Execution of this plan modifies the system by executing reconfiguration actions that deal with the activation and deactivation of software components and the creation and destruction of channels among components. For more details about the reconfiguration engine see [6].

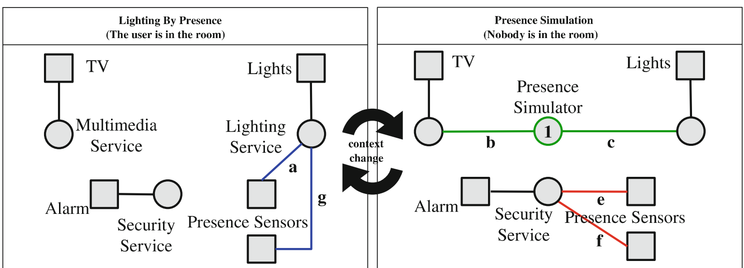


Fig. 2. Smart hotel architecture model reconfigurations

Figure 2 shows two Smart Home configurations according to the concrete syntax of the PervML. Figure 2 (left) shows a *User in the room* configuration, while Fig. 2 (right) shows a *Nobody in the room* configuration. As it can be observed, movement sensors are not used for lighting (left); instead, they are used to provide information to the security service (right). In addition, the Occupancy simulation service is activated in the *Nobody in the room* configuration, and the connections that are required for this service to communicate with multimedia, lighting, and security services are established.

3 Feature Location with Architecture Models at Run-Time

Figure 3 shows an overview of our feature location approach. In the Dynamic Analysis phase, the software engineer executes a scenario, which uses the target feature to be located. The run-time architecture model obtained from the running scenario contains the elements of the model that are related to the target feature.

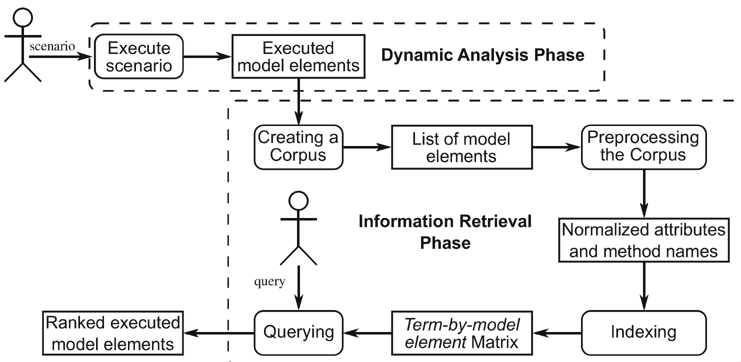


Fig. 3. Feature location approach based on architecture models at Run-Time

In the Information Retrieval phase, the approach filters the run-time architecture model to extract the relevant elements of the target feature to be located. To achieve the filtering, we adapt an Information retrieval (IR) technique named Latent Semantic Indexing (LSI) [12], which allows the software engineers to write queries that describe the feature to be located. The result is a ranked list of model elements that are related to the feature based on the similarity to the provided query.

The following subsections present the details of each one of the steps of our approach that must be carried out in order to perform the feature location at the model level. We use the Smart Hotel presented in Sect. 2 throughout the different subsections to illustrate the details with a running example.

3.1 The Dynamic Analysis Phase

Execution information is gathered via dynamic analysis (see Fig. 3), which is commonly used in program comprehension and involves executing a software system under specific conditions. Invoking the desired feature during run-time generates a feature-specific execution trace. In other words, the input for the execution is a scenario that runs the specific feature.

For example, we depict a scenario where we want to fix a bug in the gradual lights in the Smart Hotel. Therefore, the feature that we must locate is the Gradual Lighting service. We follow the information from the bug report to define the scenario that executes the targeted feature. In this case, the scenario is as follows:

'The software engineer simulates an empty Smart Hotel room. The lights are off. The software engineer simulates that a client enters the room. The lights gradually turn on. The software engineer simulates that the client leaves the room, and then the lights gradually turn off.'

3.2 The Information Retrieval Phase

Textual information in source code (represented by identifier names and internal comments) embeds domain knowledge about a software system. In our case, textual information corresponds to the names, attributes and methods of the model elements. This information can be leveraged to locate the implementation of a feature through the use of IR. IR works by comparing a set of artifacts to a query and ranking these artifacts by their relevance to the query.

There are many IR techniques that have been applied for feature location tasks. However most feature location research efforts have shown better results when LSI is applied [14,17,18]. To perform LSI, our approach follows five main steps: creating a corpus, preprocessing, indexing, querying, and generating results (see Fig. 3 Information Retrieval phase).

We adapted each step of the LSI technique to work with architecture models. Instead of using the source code files, we used the architecture model that contains the executed model elements from the dynamic analysis. The adaptation is performed as follows:

Creating a corpus. In the first step of LSI, a document granularity needs to be chosen to form a corpus. A document lists all the text found in a contiguous section of source code (methods, classes, or packages). A corpus consists of a set of documents. In this work, each document corresponds to a model element of the architecture model. Each document (model element) includes text from the names of the attributes and methods.

Preprocessing. Once the corpus is created, it is preprocessed. Preprocessing involves normalizing the text of the documents. For source code, operators and programming language keywords are removed. In addition, identifiers and compound words are split. In this work, the type of the attributes and the type of the parameters in the methods are removed. Then, all the identifiers are split; for example "IlluminationService" becomes "illumination" and "service".

Indexing. The corpus is used to create a *term-by-document matrix*. Each row of the matrix corresponds to each term in the corpus, and each column represents each document. Each cell of the matrix holds a measure of the weight or relevance of the term in the document. The weight is expressed as a simple count of the number of times that the term appears in the document. In other words, each term-document pair has a number that indicates the number of times this term appears as part of the names of attributes or methods of this model element. In this work, in the term-by-document co-occurrence matrix, the terms (rows) correspond to the names of the attributes or methods (i.e., intensity) of the run-time architecture model and the documents (columns) correspond to the model elements (i.e., IlluminationService) that have appeared in the run-time architecture model.

Figure 4 shows this term-by-document co-occurrence matrix with the values associated to our running example. Each row in the matrix stands for each one of the unique words (terms) extracted from our run-time architecture model. Figure 4 shows a set of representative keywords in the domain such as 'room',

Model Elements

Terms	Presence Service	Illumination Service	Light	Energy Service	...	Outside Detector
room	1	2	0	1	...	2
automated	2	1	0	0	...	4
light	6	5	7	0	...	0
presence	4	2	0	0	...	1
intensity	0	2	1	1	...	0
door	6	0	0	0	...	1
detector	1	1	0	2	...	2
infrared	1	1	0	2	...	1
energy	2	1	0	6	...	0
...	...	...	...	...	...	...
security	1	0	0	1	...	4

Query
1
0
2
6
1
0
0
0
0
...
0

LSI Results

Q: Query
P: Presence Service
I : Illumination Service
L: Light
E: Energy Service
O: Outside Detector

Ranking

Positions	1	2	3	...	n-1	n
Model Elements	P	I	L	...	E	O

Fig. 4. Information retrieval via Latent Semantic Indexing (LSI)

'light', or 'presence' as the terms of each row. Each column in the matrix stands for the model elements of the run-time architecture model. Figure 4 also shows the names of the model elements in the columns such as 'PresenceService' or 'IlluminationService', which represent the model elements of the run-time architecture model. Each cell in the matrix contains the frequency with which the keyword of its row appears in the document denoted by its column. For instance, in Fig. 4, the term 'light' appears 6 times in the 'PresenceService' model element.

Querying. A user formulates a query in natural language consisting of words or phrases that describe the feature to be located. Since LSI does not use a predefined grammar or vocabulary, users can originate queries in natural language. In this work, we use the bug reports to formulate the queries. Only the relevant terms are taken into account, and words such as determinants and connectors from the language are avoided.

In Fig. 4, the query column represents the words that appear in the bug report. Each cell contains the frequency with which the keyword of its row appears in the query. For instance, the term 'light' appears 2 times in the query.

Generating results. In LSI, the query and each document correspond to a vector. The cosine of the angle between the query vector and a document vector is used as the measure of the similarity of the document to the query. The closer the cosine is to 1, the more similar the document is to the query. A cosine similarity value is calculated between the query and each document, and then the documents are sorted by their similarity values. The user inspects the ranked list to determine which of the documents are relevant to the feature.

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) [14]. SVD is a form of factor analysis, or more precisely, 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 that contains scaling values such that when the three components are matrix-multiplied, the original matrix is reconstructed.

A three-dimensional graph of the LSI results is provided in Fig. 4. The graph shows the representation of each one of the vectors, labeled with letters that represent the names of the model elements, which are referenced in the box below the graph. The graph reflects the 'PresenceService' model element vector as being the closest to the query vector, followed by the 'IlluminationService' model element vector.

The goal of our approach is to rank model elements relevant to the feature to locate within the top positions. The ranking of model elements is ordered by the values of the cosines. In the running example (see Fig. 4, Ranking), the 'PresenceService' element is in the first position and therefore is the most relevant, while the 'OutsideDetector' element is in the last position and is the less relevant.

4 Evaluation: Feature Location in the Smart Hotel

We evaluated whether our feature location approach with architecture models at run-time achieves better results than current approaches [14] that use source code to perform feature location. We choose the approach presented in [14] because is the one that shows the best results for feature location in source code [8,20].

We defined the experimental design of our study using the Goal-Question-Metric method (GQM) [2]. We used the template presented in [3]. The GQM method was defined as a mechanism for defining and interpreting a set of operation goals using measurements. In this evaluation, according to GQM template our goal was the following:

- Object: Our Smart Hotel
- Purpose: Evaluation
- Issue: The accuracy of the results in our architecture-model-based feature location approach
- Context: Feature location in the run-time architecture model

To fulfill this goal, we focused on answering the following research question: Does our architecture-model-based approach for feature location provide better results than a source-code-based approach?

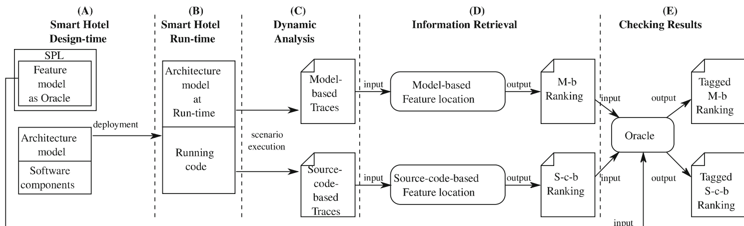


Fig. 5. The evaluation process followed in the Smart Hotel

Figure 5 shows the entire process that we followed for this evaluation.

(A) Smart Hotel Design-time. The Smart Hotel was developed using a Dynamic Software Product Line (DSPL) [4]. The architecture model and the source code of the Smart Hotel were configured with a feature model [7]. The feature model specifies the 39 different features that the Smart Hotel has implemented. We used the feature model of the software product line as an oracle to evaluate our approach. In other words, we made use of a set of PervML models and implementation codes whose feature realizations are known beforehand and will be considered as the ground truth. This enables us to compare the oracle with the results provided by our approach and the source-code approach.

The Smart Hotel presents sixty-eight model elements (thirteen services, twenty devices, and thirty-five channels) in the architecture model. The software components of the Smart Hotel consist of 268 classes that are implemented in about 67,207 methods of Java source code.

(B) Smart Hotel Run-time. In the evaluation set-up, a scale environment with real KNX[3] devices was used to represent the Smart Hotel. In our case, we chose to carry out in-virtuo experiments [2,21], where the real world is described as computer models. This experiment involves the interaction among participants and a computerized model of reality. The simulated environment offers major advantages regarding the cost and the feasibility of replicating a real-world configuration. In addition, some scenarios, such as fires or floods, cannot be replicated in the real world.

(C) Dynamic Analysis. We then ran the scenario that executes the feature to be located. For this case study, we executed 30 independent runs (as suggested by [1]) for each of the 39 features. The execution of the scenario generated the Smart Hotel run-time architecture model and source code traces.

(D) Information Retrieval. Our architecture-model-based feature location approach and the source-code-based feature location approach used the Smart Hotel run-time architecture model and source code traces, respectively. Our architecture-model-based feature location approach produced a ranking of

[3] KNX technology is a standard for applications in home and building control (http://www.knx.org/).

model elements (see Fig. 5, M-b Ranking) and the source-code-based feature location approach produced a ranking of methods (see Fig. 5, S-C-b Ranking) for the targeted feature.

(E) Checking Results. The feature model oracle enabled us to know how many of the model elements or methods in the rankings were the ones that realized the target feature. We tagged the model elements (see Fig. 5, Tagged M-b Ranking) and methods (see Fig. 5, Tagged S-C-b Ranking) that belonged to the targeted feature. This allowed us to know their positions in the rankings.

4.1 Results

We performed this evaluation with the thirty-nine features that compose the Smart Hotel. We defined the scenarios based on bug reports of each one of the features. On average, the traces generated were the following: 46 model elements in the architecture-model-based feature location approach and 3,817 methods in the source-code-based feature location approach.

Figure 6 shows the position of the first model element and the first method that belong to the target feature in the ranking for each one of the thirty-nine features. The x-axis represents the features, and the y-axis represents the position in the ranking. The blue dots represent the first model element for each feature and the red Xs represent the first source code method for each feature. The position of the first model element that belongs to each one of the features has values between 1 and 28, where the 84 % of the results are in the top ten positions. However, the position of the first source code method that belongs to each one of the features has values between 12 and 100.

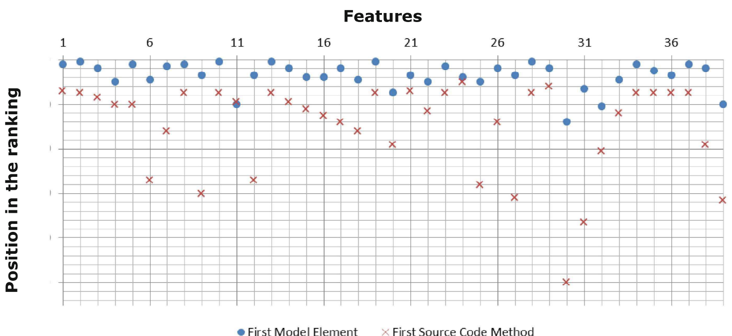


Fig. 6. Position of the first model element and the first method that belong to the target feature in the ranking for each one of the features

Does Our Architecture-model-based Approach for Feature Location Provide Better Results Than the Source-code-based Approach? Our architecture-model-based approach ranks the relevant elements in the top ten positions of the ranking in 84 % of the cases; in the top positions in the ranking of the source-code-based approach, there are false positives associated with some programming patterns and true positives are spread between positions 15 and 100 (see Fig. 6).

It is accepted by the feature location community [14,18] that, a feature location approach is considered better than another feature location when it produces a ranking where the elements that belong to the feature are in higher positions than in the ranking of the other approach. In our evaluation with the Smart Hotel, our architecture-model-based feature location approach obtained better positions in the rankings than the source-code-based approach.

4.2 Analysis of the Results

The results of our evaluation confirms that introducing architecture models at run-time outperforms the equivalent technique at source code level.

Figure 7 shows the graphical representation of the ranking for the 'Gradual Lighting' feature (feature number five in Fig. 6). Due to space constraints, we only show the graphical representation for one feature, however, all the rankings follow a similar distribution in the results.

The query is the vector that is on the x-axis. The remainder of the vectors are model elements in the architecture-model-based feature location approach or methods in the source-code-based feature location approach. Those that have been tagged by the oracle have a r_i label at the end of the arrow, while those

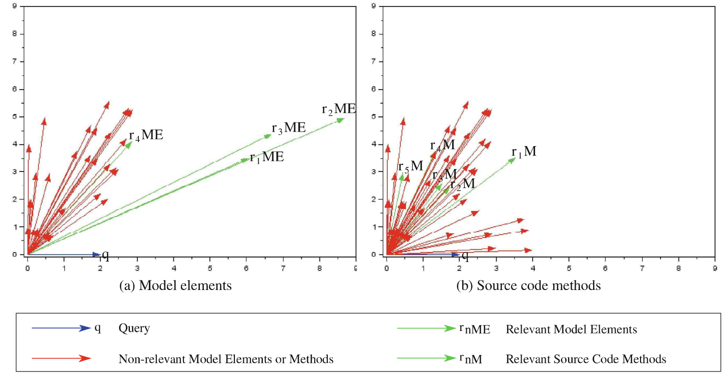


Fig. 7. Vectorial representation of the model elements and source code methods in the Ranking of the 'Gradual Lighting' feature

that have not been tagged have nothing at the end of the arrow. The angle corresponds to the cosine with which we calculated the position in the ranking (see Sect. 3.2); the closer the model element or method is to the query, the higher the position in the ranking. The length of each vector indicates the number of times that the terms appear in each model element or method. The longer the vector is, the more terms appear in the model element or method.

The graph of the left corresponds to the architecture-model-based feature location approach, of the total of vectors (model elements), forty-six, the graph only shows the thirty-three which have positive cosines, the rest, thirteen, are in the left of the y-axis and have few relevance for the query. The graph on the right corresponds to the source-code-based feature location approach, of the total of vectors (methods), 3,817, the graph only shows the 1,302, which have positive cosines, the rest, 2,515, are in the left of the y-axis.

The first difference between the architecture-model-based approach and the source-code-based approach lies in the size of the search space in which the feature must be located. The goal of a feature location technique is to reduce the effort required by software engineers to find the desired feature. Our architecture-model-based approach on average requires searching in less than fifty model elements while a source-code-based approach on average requires searching in more than three thousand eight hundred methods.

The graphical representation of Fig. 7 allows us to see that the architecture-model-based approach discriminates better than the source-code-based approach. The majority of the model elements that belong to the feature achieve better results than the ones that do not belong. However, in the source-code-based approach, the source code methods that belong to the feature and the source code methods that do not belong to the feature are not differentiated.

In addition, the vectors of the model elements that belong to the feature are closer to the query vector than the vectors of the source code methods that belong to the feature (see Fig. 7). Therefore, the model-based approach provides searches that are more accurate.

Since architecture models at run-time allow working on a high level of abstraction, the words used at the model level (i.e., *room, presence*) are closer to the query than the ones used at source code level (i.e., *save or run*). The result is that queries using a natural language show better results with the architecture-model-based approach. In the source-code-based approach some auxiliary terms are taken into account. Some terms, like *controller or run*, can proceed from some programming patterns. By raising the level of abstraction with the architecture model, we can prevent auxiliary methods and variables from interfering with the feature location.

Finally, in our Smart Hotel, we realized that the model elements that contained few attributes and methods got worse positions in the ranking than the ones that contained more attributes and methods. For example, one of the elements related to the feature 'Gradual Lighting' in Fig. 7 obtained position 27 in the ranking. This is because this element corresponds to a channel element that connects two services. This particular channel only has three attributes

that describe the information that goes through the channel. The information required by this element was not as detailed as the other model elements when specifying the model. For this reason, the model element corresponding to this channel got a lower position in the ranking. In contrast, other kinds of channels got better positions since, on average, they have about twenty attributes and methods.

4.3 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 generalizations of the results.

One issue is whether or not the software system used in the evaluation is representative of those used in practice. Given the scale and complexity of our Smart Hotel (sixty-eight model elements and 67.207 methods), we consider our evaluation to be a good starting point for representing a realistic case. However, this threat can be reduced if we experiment with other software systems of different sizes and domains.

Furthermore, the DSL model used in this study is a language in a specific domain. PervML is a DSL that describes pervasive systems using high-level abstraction concepts. However, other experiments with other DSLs should be performed to validate our findings.

Another issue is the selection of the scenarios based on the bug reports to obtain the execution trace. Since we are experts in the Smart Hotel system, we can claim that our scenarios are good representatives of features that have been necessary to locate in order to solve the most common bugs of the Smart Hotel. Thus, depending on the chosen scenarios, the results may differ. The more knowledge the software engineer has about the system, the better the scenarios and the queries will be, leading to better results.

5 Related Work

Some approaches that are related to feature location use design-time models to extract variability. Although they do not use architecture models at run-time, their works are based on extracting features using models.

Font et al. [9] suggest that model fragments that are extracted mechanically may not be recognizable units by the application engineers. They propose identifying model patterns by human-in-the-loop and conceptualizing them as reusable model fragments. Their approach provides the means to identify and extract those model patterns and further apply them to existing product models. In [10], the work from [9] is extended to handle situations where the domain expert fails to provide accurate information. The authors propose a genetic algorithm for feature location in model-based software product lines. When this method was compared with another approach that did not use a genetic algorithm, the results showed that their approach was able to provide solutions for situations where the information of the domain expert was inaccurate, while the other approach failed.

Martinez et al. [15] propose an extensible framework that allows features to be identified, located, and extracted from a family of models. They introduce the principles of this framework and provide insights on how it can be extended for use it in different scenarios. As a result, the initial investment required by the task of adopting a software product line from a family of models is reduced.

Xue et al. [22] present an approach to support effective feature location in products variants. They exploit commonalities and differences of product variants by software differencing and formal concept analysis (FCA) techniques so that IR techniques can achieve satisfactory results.

All of these works are based on extracting model fragments from a given set of models taking into account their commonalities and variabilities. However, these approaches do not take into account the run-time behaviour of the systems and are not focused on feature location. Nevertheless, all of them can be used as a basis for the extraction of the model fragments that correspond to the feature to be located.

There are many more research efforts in dynamic feature location techniques that are based on source code analysis. Some of these works combine other kinds of analysis (i.e., information retrieval) to obtain more accurate results.

Liu et al. [14] combine information from an execution trace and from the comments and identifiers from the source code. They executed a single scenario (which runs the desired feature), and all executed methods are identified based on the collected trace using LSI. The result is a ranked list of executed methods based on their textual similarity to a query. Similarly, Koschke et al. [11] develop a semi-automated technique using a combination of static and dynamic program analysis. However, they use FCA to explore the results of the dynamic analysis.

Revelle et al. [18] apply data fusion for feature location. Their technique combines information from textual, dynamic, and web mining analysis applied to a software system. Their input is a single scenario that executes the feature; after running the scenario, they construct a call graph that contains only the methods that were executed. Then, they apply a web-mining algorithm, and the system filters out low-ranked methods. The remaining set of methods is scored using LSI based on their relevance to the input query that describes the feature.

Similarly to our approach, all these feature location techniques use information from different sources. Additionally, Revelle and Poshyvanyk [19] present an exploratory study of feature location techniques that use various combinations of textual, dynamic, and static analysis. Also, they introduces a new way of applying textual analysis by which queries are automatically composed by identifiers of a method known to be relevant to a feature. Although they are based on locating feature in source code, some of the ideas could be applied to our architecture-model-based feature location approach to obtain more accurate results.

6 Conclusions

This work proposes an approach that combines architecture models at run-time and information retrieval for feature location. Specifically, our approach uses a

scenario that executes the desired feature to be located. In addition, our approach ranks all of the model elements that are executed to extract the model elements that are related to the feature. We adapt an information retrieval technique called LSI to work with architecture models at run-time. The ranked list of model elements is obtained based on the similarity of these model elements to a query in a natural language.

Both models and feature descriptions are on a higher abstraction level than source code. This means that models are closer to natural language queries, and the results are more accurate. The comparison of our architecture-model-based feature location approach with a source-code-based feature location approach for the Smart Hotel case study demonstrate this outcome.

Our architecture-model-based approach ranks the relevant elements in the top ten positions of the ranking in 84 % of the cases. In the top positions of the source-code-based approach ranking, there are false positives associated with some programming patterns and true positives are spread between positions 12 and 100.

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

References

1. Arcuri, A., Briand, L.: A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. Softw. Test. Verif. Reliab. **24**(3), 219–250 (2014). doi:10.1002/stvr.1486
2. Basili, V.R.: The role of experimentation in software engineering: past, current, and future. In: Proceedings of the 18th International Conference on Software Engineering, ICSE 1996, pp. 442–449 (1996). http://dl.acm.org/citation.cfm?id=227726.227818
3. Basili, V.R., Caldiera, G., Rombach, H.D.: The goal question metric approach. In: Calvo, J. (ed.) Encyclopedia of Software Engineering. Wiley, Hoboken (1994)
4. Bencomo, N., Hallsteinsen, S., Santana de Almeida, E.: A view of the dynamic software product line landscape. Computer **45**(10), 36–41 (2012)
5. Bencomo, N., France, R., Cheng, B.H.C., Aßmann, U. (eds.): Models@run.time. Foundations, Applications, and Roadmaps. LNCS, vol. 8378. Springer, Heidelberg (2014)
6. Cetina, C.: Achieving autonomic computing through the use of variability models at run-time. Ph.D. thesis, Universidad Politécnica de Valencia (2010)
7. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged configuration using feature models. In: Nord, R.L. (ed.) SPLC 2004. LNCS, vol. 3154, pp. 266–283. Springer, Heidelberg (2004). doi:10.1007/978-3-540-28630-1_17
8. Dit, B., Revelle, M., Gethers, M., Poshyvanyk, D.: Feature location in source code: a taxonomy and survey. J. Softw. Maintenance Evol. Res. Pract. **25**(1), 53–95 (2011)

9. Font, J., Arcega, L., Haugen, Ø., Cetina, C.: Building software product lines from conceptualized model patterns. In: Proceedings of the 2015 19th International Software Product Line Conference, SPLC 2015, Nashville, TN, USA (2015)
10. Font, J., Arcega, L., Haugen, Ø., Cetina, C.: Feature location in model-based software product lines through a genetic algorithm. In: Kapitsaki, G.M., Santana de Almeida, E. (eds.) ICSR 2016. LNCS, vol. 9679, pp. 39–54. Springer, Heidelberg (2016). doi:10.1007/978-3-319-35122-3_3
11. Koschke, R., Quante, J.: On dynamic feature location. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, ASE 2005, NY, USA, pp. 86–95 (2005). http://doi.acm.org/10.1145/1101908.1101923
12. Landauer, T.K., Foltz, P.W., Laham, D.: An introduction to latent semantic analysis. Discourse Process. **25**(2–3), 259–284 (1998)
13. Lehman, M.M., Ramil, J., Kahen, G.: A paradigm for the behavioural modelling of software processes using system dynamics. Technical report, Imperial College of Science, Technology and Medicine, Department of Computing, September 2001
14. Liu, D., Marcus, A., Poshyvanyk, D., Rajlich, V.: Feature location via information retrieval based filtering of a single scenario execution trace. In: Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering, ASE 2007, NY, USA, pp. 234–243 (2007). http://doi.acm.org/10.1145/1321631.1321667
15. Martinez, J., Ziadi, T., Bissyandé, T.F., Le Traon, Y.: Bottom-up adoption of software product lines: a generic and extensible approach. In: Proceedings of the 2015 19th International Software Product Line Conference, SPLC 2015, Nashville, TN, USA (2015)
16. Muñoz, J.: Model driven development of pervasive systems. building a software factory. Ph.D. thesis, Universidad Politécnica de Valencia (2008)
17. Poshyvanyk, D., Gueheneuc, Y.G., Marcus, A., Antoniol, G., Rajlich, V.: Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. IEEE Trans. Softw. Eng. **33**(6), 420–432 (2007). doi:10.1109/TSE.2007.1016
18. Revelle, M., Dit, B., Poshyvanyk, D.: Using data fusion and web mining to support feature location in software. In: 2010 IEEE 18th International Conference on Program Comprehension (ICPC), pp. 14–23, June 2010
19. Revelle, M., Poshyvanyk, D.: An exploratory study on assessing feature location techniques. In: IEEE 17th International Conference on Program Comprehension, ICPC 2009, pp. 218–222, May 2009
20. Rubin, J., Chechik, M.: A survey of feature location techniques. In: Reinhartz-Berger, I., Sturm, A., Clark, T., Cohen, S., Bettin, J. (eds.) Domain Engineering, pp. 29–58. Springer, Berlin (2013)
21. Travassos, M.O., Barros, M.O.: Contributions of in virtuo and in silico experiments for the future of empirical studies in software engineering. In: Proceedings of the ESEIW 2003 Workshop on Empirical Studies in Software Engineering. IEEE Computer Society (2003)
22. Xue, Y., Xing, Z., Jarzabek, S.: Feature location in a collection of product variants. In: 2012 19th Working Conference on Reverse Engineering, pp. 145–154, October 2012

12.4 ECMFA'17 Paper

Title: On the Influence of Models at Run-Time Traces in Dynamic Feature Location.

Authors: Lorena Arcega, Jaime Font, Øystein Haugen, Carlos Cetina.

Proceedings: Proceedings of the 13th European Conference on Modelling Foundations and Applications (ECMFA '17).

Location: Marburg, Germany - July 19-20, 2017

Publisher: Springer

Pages: 90-105

DOI: `https://doi.org/10.1007/978-3-319-61482-3_6`

Contribution: Lorena Arcega is the main author of the paper and is responsible for 90% of the work. She was also responsible for the oral presentation of the work which took place during the conference.

On the Influence of Models at Run-Time Traces in Dynamic Feature Location

Lorena Arcega[1,2(✉)], Jaime Font[1,2], Øystein Haugen[3], and Carlos Cetina[1]

[1] SVIT Research Group, Universidad San Jorge, Zaragoza, Spain
{larcega,jfont,ccetina}@usj.es
[2] Department of Informatics, University of Oslo, Oslo, Norway
[3] Department of Information Technology, Østfold University College, Halden, Norway
oystein.haugen@hiof.no

Abstract. Feature Location is one of the most important and common activities performed by developers during software maintenance and evolution. In prior work, we show that Dynamic Feature Location obtains better results working with models rather than source code. In this work, we analyze how the criteria to create the model traces influence the Dynamic Feature Location results. We distinguish between two different criteria: configuration and architecture. Our Dynamic Feature Location approach is composed of dynamic analysis, information retrieval at the model trace level, and information retrieval at the model level. The evaluation in a Smart Hotel tests whether the traces created following the two criteria modify the results of the Feature Location by measuring recall, precision, and the combination of both (F-measure). The results reveal that in 75% of the cases the traces that follow the architecture criterion outperform the traces that follow the configuration criterion.

Keywords: Models at run-time · Feature location · Reverse engineering

1 Introduction

Software maintenance often involves tedious, time-consuming activities. Lehman et al. [15] pointed out that up to 80% of the lifetime of a system is spent on maintenance and evolution activities. Software maintainers spend from 50% to almost 90% of their time trying to understand a program to make changes correctly. To understand the underlying intents of an unfamiliar system, maintainers look for clues in both the code and the documentation [2].

Feature Location is one of the most important and common activities performed by developers during software maintenance and evolution [8]. Currently, research efforts in Feature Location are concerned with identifying software artifacts that are associated with a program functionality (a feature). In Feature Location approaches, it is common to focus on analyzing source code.

In prior work [3] we show that, for systems based on models at run-time, better results were obtained in Dynamic Feature Location if we analyzed the

© Springer International Publishing AG 2017
A. Anjorin and H. Espinoza (Eds.): ECMFA 2017, LNCS 10376, pp. 90–105, 2017.
DOI: 10.1007/978-3-319-61482-3_6

run-time model instead of the source code. Through this work, our goal is to analyze how the criteria to form the model trace influence the Dynamic Feature Location results. We are interested in two criteria to decide when a snapshot of the run-time model should be added to the trace: (1) configuration criterion, that adds a snapshot of the run-time model to the trace when the model corresponds to a target configuration of the system in a reconfiguration, and (2) architecture criterion that adds a snapshot of the run-time model to the trace each time a change in the run-time model is performed.

Our Dynamic Feature Location approach is composed of dynamic analysis, information retrieval at the model trace level, and information retrieval at the model level. As a result, our approach generates a ranking with the most relevant model elements for the feature to be located. We implemented the second and third steps using a method named Latent Semantic Indexing (LSI), the method that provides better results [16,20,21]. LSI allows software engineers to write queries that are relevant to the feature they want to locate. As a result, the software engineers obtain a ranked list of model elements from the model, which are intended to identify the parts of the model that are significant for the target feature.

We have applied our approach to a Smart Hotel to assess its performance. The case study presents 476 model elements in the architecture model. The evaluation tests how the traces created following the two criteria influence the results of the Feature Location by measuring recall, precision, and the combination of both (F-measure). These are the most common measures for the experiments with information retrieval methods [17,23]. The recall, precision, and F-measure values reveal that the traces that follow the architecture criterion obtain better results than the traces that follow the configuration criterion in 75% of the cases.

The remainder of the paper is structured as follows. In Sect. 2, we present the Smart Hotel and the model traces. In Sect. 3, we describe our approach for Dynamic Feature Location with models. In Sect. 4, we evaluate our approach in the Smart Hotel and we discuss the results. In Sect. 5, we examine the related work of the area. Finally, we present our conclusions in Sect. 6.

2 Background

The running example and the evaluation of this paper are performed through a Smart Hotel [7]. In this section we present the reconfigurations of the Smart Hotel that are performed in response to changes in the context. For instance, a change in the context could be determined by assessing if there is a client in the room or not, or focusing on what activities the client may be performing (sleeping, watching TV, etc.). In addition, this section shows the model traces in which our approach records the execution information.

2.1 Behavior of the Smart Hotel at Run-Time

The Smart Hotel reconfiguration engine determines how the system should be reconfigured in response to a context change, and then it modifies the

architecture model accordingly. In models at run-time, a causal connection between the system and the run-time model is defined (there is a bidirectional relation between the source code and the run-time model). This connection allows the models (usually the architecture model) to reflect the software state. This connection can be achieved in different ways, however, the most used implementation is the MAPE-K loop [6,13]. For more details about the reconfiguration engine of the Smart Hotel see [7].

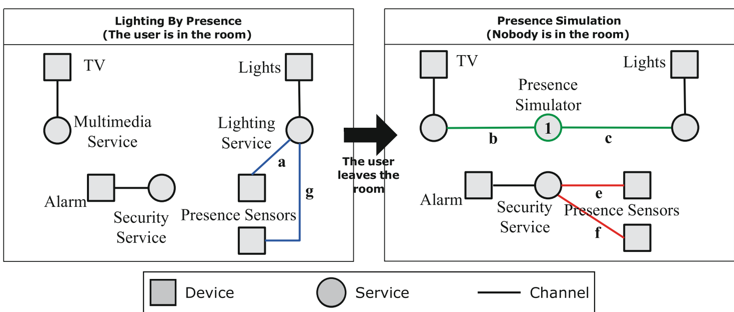


Fig. 1. Smart hotel model reconfigurations

Figure 1 shows two Smart Hotel configurations according to the concrete syntax of the architecture model of PervML [19]. Figure 1 (left) shows a *User in the room* configuration, while Fig. 1 (right) shows a *Nobody in the room* configuration. It can be observed that movement sensors are used for different purposes: lighting (left), and providing information to the security service (right). In addition, the Occupancy simulation service is activated in the *Nobody in the room* configuration, and the connections that are required for this service to communicate with multimedia, lighting, and security services are established.

2.2 Model Execution Traces

In our approach, the execution information is recorded by a model trace of snapshots at run-time. Each execution trace is related to a set of snapshots of the run-time model. In this paper we are interested in two criteria to decide when a snapshot of the run-time model should be added to the trace: (1) configuration criterion, and (2) architecture criterion.

In the configuration criterion, the snapshots are added to the trace when the run-time model corresponds to a target configuration of the system in a reconfiguration. That is, a snapshot is added when the system completes the changes from one configuration to another. In the architecture criterion, the snapshots are added to the trace when a change in the run-time model is performed. That is, a snapshot is added each time a component of the run-time model is deleted

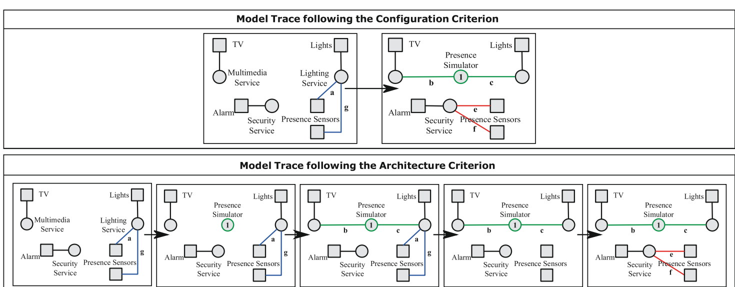


Fig. 2. Different model traces following the different criterion

or created even if the model does not correspond to a target configuration of the system.

Figure 2 shows two different traces for the same reconfiguration (the reconfiguration showed in Fig. 1). The upper part shows a trace composed by the configuration criterion. The first snapshot shows the system when there is a user in the room, and the second snapshot shows the system when the user leaves the room and the corresponding reconfiguration is completed. The bottom part shows a trace composed by the architecture criterion. The first snapshot and the last one are the same, as in the upper part of the figure. However, the rest of the snapshots give more detail on what actions were carried out in the reconfiguration from the first snapshot to the fifth one. For instance, in the second snapshot, the *Presence Simulator* appears; in the third snapshot, the channels that connect the *Presence Simulator* with the *Multimedia Service* and the *Lighting Service* emerge; in the fourth snapshot, the channels that connect the *Lighting Service* with the *Presence Sensors* are deleted; and, finally, in the fifth snapshot, the channels that connect the *Security Service* with the *Presence Sensors* come into sight.

3 Model Based Dynamic Feature Location Approach

Figure 3 shows an overview of our model based Dynamic Feature Location approach. It is composed of three steps: dynamic analysis, information retrieval in the model trace, and information retrieval in a model from the model trace. In the first step, the software engineer executes a scenario, which involves the desired feature to be located. The execution information is recorded by a model trace of snapshots at run-time. Then, the model trace is used as input for the second step of our Dynamic Feature Location. Using information retrieval, the most relevant model for the desired feature is selected from the model trace. This model is used as input for the third step of our approach, which performs information retrieval at the model element level. As a result, the software engineers obtain a ranked list of model elements from the model, intended to identify the parts of the model that are significant for the target feature.

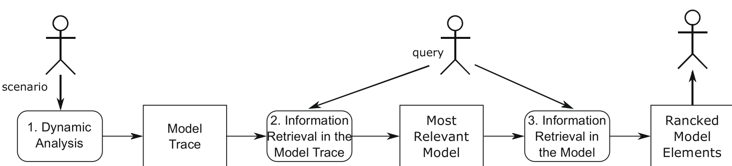


Fig. 3. Overview of the dynamic feature location approach

The following subsections present each of the steps that must be carried out in order to perform the Feature Location at the model level, following our approach. We use the Smart Hotel presented in Sect. 2 throughout the different subsections to illustrate the details with a running example.

3.1 Dynamic Analysis

Execution information is gathered via dynamic analysis, which is commonly used in program comprehension and involves executing a software system under specific conditions. Executing the target feature during run-time generates a feature-specific execution trace. In other words, the input for the execution is a scenario that runs the target feature.

The model trace generated in this step only includes the models that have been executed in the feature-specific scenario. This model trace is the main artifact that our approach uses to locate the target feature.

As an example, we depict a scenario where we want to fix a bug in the gradual light of the Smart Hotel. We follow the information from the bug report to define the scenario that executes the target feature. In this case, a simplified version (due to space constraints) of the scenario is as follows:

'The software engineer simulates an empty Smart Hotel room. The lights are off. The software engineer simulates that a client enters the room. The lights gradually turn on. The software engineer simulates that the client leaves the room, and then the lights gradually turn off.'

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

3.2 Information Retrieval in the Model Trace

In this step we use the model trace extracted in the previous step. In addition, the software engineer has to formulate a query related to the feature that must be located. The model trace and the query can be leveraged to locate the most relevant model for the feature through the use of Information Retrieval (IR). IR works by comparing a set of artifacts to a query, and ranking these artifacts by their relevance to the query.

Typically, the query can come from textual documentation of the products, comments in the code, bug reports or oral descriptions from the engineers. Therefore, the query will include some domain specific terms similar to those used when specifying the models. The knowledge of the engineers about the domain and the models will be useful to select the query from the available sources.

There are many IR techniques that have been applied for feature location tasks. Most of the feature location research efforts show better results when applying Latent Semantic Indexing (LSI) [16,21]. In addition, combining LSI with dynamic analysis improves its effectiveness [20].

In our previous work [3] we adapted LSI, which was traditionally used in code, in order to apply it to models. Summarizing, the text from the models is extracted and a text corpus is created, where each document corresponds to a model or to a subset of model elements of the model. The text corpus is used to create a term-by-document co-occurrence matrix. As LSI does not use a predefined grammar or vocabulary it is very robust regarding outlandish identifier names and stop words. Users can produce queries in natural language and the system returns a list of all the documents in the system, ranked by their semantic similarity to the query.

Terms	Models: Snapshot 1	Snapshot 2	Snapshot 3	Snapshot 4	Snapshot 5	Query
room	1	2	0	1	2	1
automated	2	1	0	0	4	0
light	6	5	7	0	0	2
presence	4	2	0	0	1	6
intensity	0	2	1	1	0	1
...	...	...	...	...	...	...
security	1	0	0	1	4	0

Fig. 4. Information retrieval via latent semantic indexing (LSI)

We adapt each step of the LSI technique to work with the model trace. The adaptation is performed as follows:

– **Creating a corpus:** Each document corresponds to a model of the model trace extracted in the dynamic analysis. Each document (model) includes text from the names of the model elements and the names of the attributes and methods of the model elements that compose that model.

- **Preprocessing:** The type of the attributes and the type of the parameters in the methods are removed. Then, all the identifiers are split. For example, 'IlluminationService' becomes 'illumination' and 'service'. To do this, we apply Natural Language Processing (NLP) techniques, such as tokenizing, Parts-of-Speech (POS) tagging techniques, and stemming techniques [1,12], however, the details of the application of these techniques are out of the scope of this paper.
- **Indexing:** In the term-by-document co-occurrence matrix, the terms (rows) correspond to the names of the model elements and the names of the attributes or methods of the model, and the documents (columns) correspond to the models that have appeared in the model trace. Figure 4 shows the term-by-document co-occurrence matrix, with the values associated to our running example.
  Each row in the matrix stands for each one of the unique words (terms) extracted from our models. Figure 4 shows a set of representative keywords in the domain such as 'room', 'light', or 'presence' as the terms of each row. Each column in the matrix stands for the models of the model trace. Figure 4 shows the models of the trace in each column, such as 'Snapshot1', which represents the first model of the model trace.
  Each cell in the matrix contains the frequency with which the keyword of its row appears in the document denoted by its column. For instance, in Fig. 4, the term 'light' appears 6 times in the 'Snapshot1' model.
- **Querying:** We use the bug reports to formulate the queries. Only the relevant terms are taken into account, and words such as determinants and connectors from the language are omitted.
  In Fig. 4, the query column represents the words that appear in the bug report. Each cell contains the frequency with which the keyword of its row appears in the query. For instance, the term 'presence' appears 6 times in the query.
- **Generating results:** In our approach, each document and the query are translated into vectors. The cosine of the angle between the query vector and a document vector is used as a measure of the similarity of the document to the query. The closer the cosine is to one, the more similar the document is to the query. A cosine similarity value is calculated between the query and each document, and then the documents are sorted according to their similarity values. The user inspects the ranked list to decide which of the documents are relevant to the feature.
  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) [14]. 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 this step of our approach, we only take into account the model that presents the best similarity measure. We consider it as the most relevant model for the feature to be located, and as such, it is used as input for the next step.

3.3 Information Retrieval in the Model

In this step we apply LSI at the model element level, considering that each model element is a document. We apply it to the model obtained in the previous step. This model is the most relevant model for the desired feature. However, we want to locate the most relevant model elements for the desired feature. The result of this step is a ranked list of model elements of the model, which are intended to identify the parts of the model that are significant for the target feature.

To that extent, we adapted LSI to work with a model. The main differences from the previous adaptation are the following:

- The input is one model. As such, the terms are extracted taking into account only one model.
- The granularity of the corpus changes. In the corpus creation, each document corresponds to a model element of the most relevant model extracted before.

For generating the results, we apply the same technique as in the previous step (SVD). However, the result in which we are interested is different. In this step of our approach, of all the model elements, only those model elements that have a similarity measure greater than x must be taken into account to measure the quality of the results. A good heuristic that is widely used is $x = 0.7$. This value corresponds to a 45° angle between the corresponding vectors. This threshold has yielded good results in other similar works [17,22]. Determining a more generally usable heuristic for the selection of the appropriate threshold is an issue under study, over which further research is needed.

The goal of our approach is to rank the relevant model elements within the top positions. The ranking of model elements is ordered by the values of the cosines.

4 Evaluation: Feature Location in the Smart Hotel

We evaluate how the architecture changes recorded with the snapshots in the model trace influence the results of Feature Location. In other words, we want to evaluate whether all the changes produced in the architecture model when a system reconfiguration is necessary are relevant for feature location. In order to do this, we compare the presented model based Dynamic Feature Location approach using traces following the architecture criterion (DFL-AT), against the same approach using traces following the configuration criterion (DFL-CT).

The quality of the results of Information Retrieval techniques is measured by their recall and precision. These are two of the most common measures for experiments with information retrieval methods [17,23]. For a given query, recall is the percentage of retrieved documents that are relevant to the total number of relevant documents, while precision is the percentage of the retrieved documents that are relevant to the total number of retrieved documents. A measure that combines both recall and precision is the harmonic mean of precision and recall, called the F-measure.

We defined the experimental design of our study using the Goal-Question-Metric method (GQM) [4]. We used the template presented in [5]. The GQM method was defined as a mechanism for defining and interpreting a set of operation goals using measurements. In this evaluation, the object is our Smart Hotel, the purpose is evaluation, the issues are the recall and precision of our Dynamic Feature Location approach, and the context is Feature Location using model traces. We focused on answering this research question: Do the criteria used to form the model trace influence the results of Dynamic Feature Location?

Basili in [4] and Travassos in [24] describe four kinds of studies: in-vivo, in-vitro, in-virtuo, and in-silico. In our case, we chose to carry out in-virtuo experiments, where the real world is described through computer models. This experiment involves the interaction among participants and a computerized model of reality. The simulated environment offers major advantages regarding cost and feasibility against replicating a real-world configuration. In addition, some scenarios such as fires or floods that cannot be replicated in the real world can be described and analyzed in a simulated environment.

In order to evaluate the results of our experiments, we have collected the existing documentation about the bugs in the Smart Hotel. Each bug can be mapped to a subset of model elements of a model, specified with the model fragment formalization capacities of the Common Variability Language (CVL). In other words, for each bug, we know beforehand which is the associated subset of model elements that are involved in the bug. We use the existing knowledge as an oracle to evaluate the results provided by DFL-AT and DFL-CT.

Figure 5 shows the entire process that we followed to evaluate our approach. For the evaluation, we used the Smart Hotel system (Fig. 5(A)). The Smart Hotel

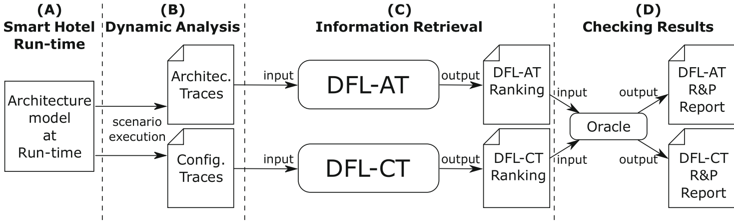


Fig. 5. Overview of the evaluation process

presents 476 model elements in the architecture model. In the evaluation set-up, a simulated environment was used to represent the Smart Hotel.

After running the scenario that executes the feature to be located, our approach generated the model traces (Fig. 5(B)). Then, we run two different Feature Location scenarios, using different traces as input. DFL-AT used the model trace that follows the architecture criterion, and DFL-CT used the model trace that follows the configuration criterion.

DFL-AT produced a ranking of model elements (DFL-AT Ranking), and DFL-CT produced another ranking of model elements (DFL-CT Ranking) for the desired feature (see Fig. 5(D)). The oracle allowed us to know how many of the model elements in the rankings were the ones that realized the desired feature in terms of recall, precision, and F-measure values.

The recall and precision were calculated as follows:

$$Recall = \frac{RankingElements \cap OracleElements}{OracleElements}$$

$$Precision = \frac{RankingElements \cap OracleElements}{RankingElements}$$

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

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

4.1 Results

We performed this evaluation with thirty bugs extracted from the documentation of the Smart Hotel. We defined the scenarios based on bug reports. On average, the generated traces were as follows: 26 models in the trace following the architecture criterion (DFL-AT) and 9 models in the trace following the configuration criterion (DFL-CT).

Figure 6 shows the recall, precision, and F-measure values for each one of the bugs. On average, DFL-AT obtains a 74.67% recall value while DFL-CT obtains a 64.23% recall value. The values indicate that around the 75% of the model elements that realize the target feature are retrieved. DFL-AT improves the recall result achieved by DFL-CT by around 10%.

Regarding the precision value, on average, DFL-AT obtains a 75.96% while DFL-CT obtains a 65.53%. The values indicate that around the 76% of the model elements retrieved belong to the targeted feature. Once again, DFL-AT improves the precision result achieved by DFL-CT by around 10%.

Consequently, on average, DFL-AT obtains a 74.35% F-measure value, while DFL-CT obtains a 63.02% F-measure value. In 75% of the cases, DFL-AT outperforms the results of DFL-CT.

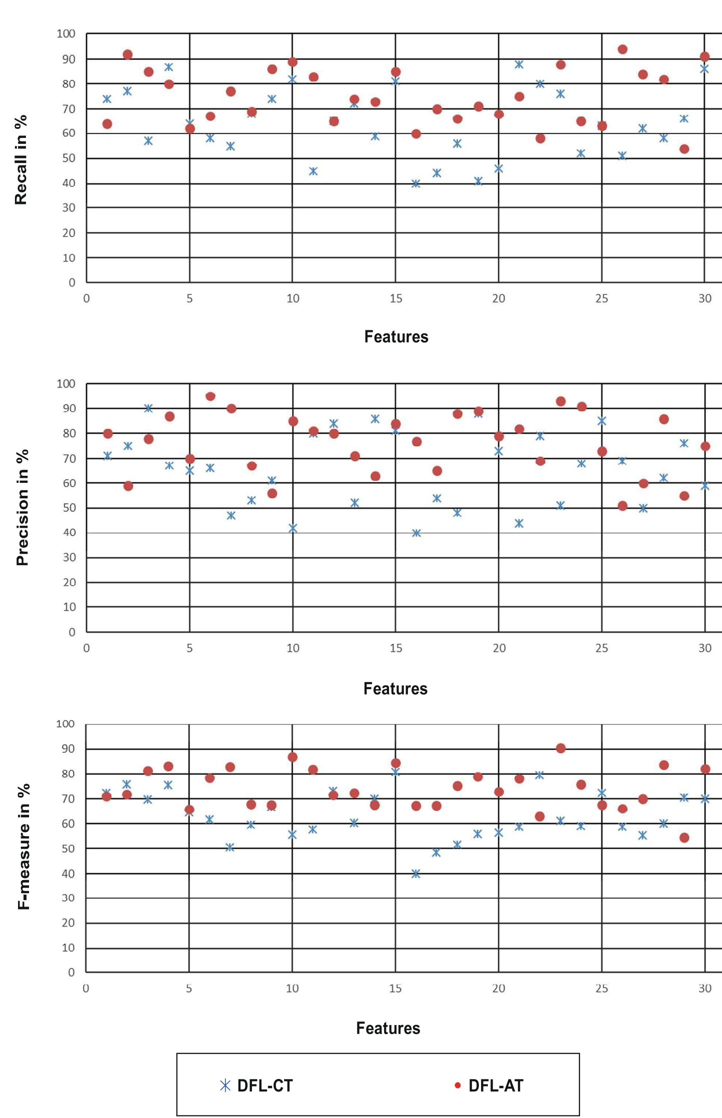


Fig. 6. Recall, precision and F-measure graphs

4.2 Discussion

Our evaluation suggests that Feature Location with model traces following the architecture criterion obtains better results in precision, recall, and F-measure than Feature Location with model traces following the configuration criterion. This is because the manifestation of a bug can occur in a snapshot that does not represent a source or target configuration in a reconfiguration of the system. In other words, a bug can be introduced in the system due to changes made in the architecture model during a reconfiguration.

Analyzing the results, Dynamic Feature Location with model traces following the architecture criterion does not always get the best results. The model traces composed by the architecture criterion have more snapshots that the model traces composed by the configuration criterion (see Sect. 2.2). In addition, two consecutive snapshots of the model trace composed by the architecture criterion are typically more similar that two consecutive snapshots of the model trace composed by the configuration criterion. For instance, in the model trace composed by the architecture criterion, a snapshot may differ from its consecutive one on only a single channel.

The above indicates that, in the step through which we perform information retrieval to extract the most representative model, the search space is larger in the model trace composed by the architecture criterion. In addition, the fact that the models in the trace are similar can imply similarity of terms in the documents of the LSI, therefore causing the technique to not discriminate between some models. However, in 75% of the cases, the Dynamic Feature Location with the model trace composed by the architecture criterion obtains better results than the Dynamic Feature Location with the model trace composed by the configuration criterion.

Finally, when forming the traces in the architecture criterion, only the creation and deletion of model elements are taken into account. In order to obtain better results in Feature Location, further experiments must be performed to analyze if other updates in the model elements should be taken into account.

4.3 Threats to Validity

In this section, we discuss some of the issues that might have affected the results of the evaluation and that may limit the generalizations of the results.

The first issue is regarding whether or not the software system used in the evaluation is representative of those used in practice. Given the scale and complexity of our Smart Hotel, we consider our evaluation to be a good starting point for representing a realistic case. However, this threat can be reduced if we experiment with other software systems of different sizes and domains.

Another issue is the selection of the scenarios to obtain the execution trace. Since we have extracted the information from the bug reports, we can claim that our scenarios are good representatives of features that must be located to solve the most common bugs of the Smart Hotel. In addition, following the

information from the bug anyone could define the scenarios. However, depending on the chosen scenarios, the results may differ.

Since the queries formulated to generate the ranked lists depend on the bug reports, the final results are also sensitive to the queries extracted by the software engineers from the bug reports.

5 Related Work

Some approaches related to Feature Location use design-time models to extract variability. Although they do not use models at run-time, their works are based on extracting features using models.

Font et al. [10] show that model fragments extracted mechanically may not be units recognizable by application engineers. They propose identifying model patterns by their human-in-the-loop approach, and conceptualizing them as reusable model fragments. Their approach provides the means to identify and extract those model patterns and further apply them to existing product models. In [11], the work from [10] is extended to handle situations where the domain expert fails to provide accurate information. The authors propose a genetic algorithm for feature location in model-based SPLs. Their comparison with other approach without a genetic algorithm demonstrates that their approach is able to provide solutions upon inaccurate information on the part of the domain expert while the other fails.

Martinez et al. [18] propose an extensible framework that allows a feature to be identified, located and extracted from a family of models. They introduce the principles of this framework and provide insights on how it can be extended for usage in different scenarios. As a result, the initial investment required by the task of adopting a software product line from a family of models is reduced.

All of these works extract model fragments from a given set of models, taking into account their commonalities and variabilities. However, these approaches do not take into account the run-time behavior of systems, and are not focused on Feature Location. Nevertheless, all of them can be used as a base for extracting the model fragments that correspond to the feature to be located.

There are many more research efforts in Dynamic Feature Location techniques based on source-code analysis. Some of these works combine other kinds of analysis (i.e. information retrieval) to obtain more accurate results.

Liu et al. [16] combine information from an execution trace and from the comments and identifiers from the source code. They executed a single scenario which executes the desired feature. All the executed methods are identified based on the collected trace using LSI. The result is a ranked list of executed methods based on their textual similarity to a query.

Revelle et al. [21] apply data fusion for feature location. Their technique combines information from textual, dynamic, and web mining analysis applied to software. Their input is a single scenario that exercises the feature. After running the scenario, they construct a call graph that contains only the methods that were executed. Then, they apply a web-mining algorithm, and the system

filters out low-ranked methods. The remaining set of methods is scored using LSI based on their relevance to the input query describing the feature.

Dit et al. [9] present a data fusion model for feature location that is based on the idea that combining data from several sources in the right proportions will be effective at identifying a feature's source code. The data fusion model defines different types of information that can be integrated to perform feature location including textual, execution, and dependence. Textual information is analyzed by IR, execution information is collected by dynamic analysis, and dependencies are analyzed using link analysis algorithms.

Similarly to our technique, all of these feature location techniques use information from different sources. Although they are based on locating features in source code, some of the ideas could be applied to our model based Dynamic Feature Location approach to obtain more accurate results.

In addition, Arcega et al. [3] present a model-based feature location approach. They apply dynamic analysis and information retrieval with run-time models. The evaluation is focused in revealing that model based feature location approaches provide more accurate results. This work extends this approach changing the way the model traces are treated. Through this work, we are focused in finding the information needed in the model traces to obtain more accurate results in Dynamic Feature Location.

6 Conclusion

In the presented work, we analyze how the criteria to create the model traces influence Dynamic Feature Location results. We focus on two different criteria: (1) configuration criterion, that adds a snapshot of the run-time model to the trace when the model corresponds to a target configuration of the system in a reconfiguration, and (2) architecture criterion, that adds a snapshot of the run-time model to the trace each time a change in the run-time model is performed. Our Dynamic Feature Location approach is composed by dynamic analysis, information retrieval at the model trace level, and information retrieval at the model level.

Our evaluation in a Smart Hotel calculates the values of the most common measures for experiments with information retrieval methods (recall, precision, and F-measure). We use these values to compare Dynamic Feature Location with traces created following the architecture criterion against Dynamic Feature Location with traces created following the configuration criterion. The results reveal that in 75% of the cases, Dynamic Feature Location with model traces composed by the architecture criterion obtains better results than Dynamic Feature Location with model traces composed by the configuration criterion.

Our future work involves designing a Feature Location approach that combines model traces and information about the time of the execution. In addition, further experiments are necessary to identify other different criteria to create model traces.

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

References

1. Alves, V., Schwanninger, C., Barbosa, L., Rashid, A., Sawyer, P., Rayson, P., Pohl, C., Rummler. A.: An exploratory study of information retrieval techniques in domain analysis. In: 2008 12th International Software Product Line Conference, pp. 67–76, September 2008
2. Antoniol, G., Gueheneuc, Y.-G.: Feature identification: an epidemiological metaphor. IEEE Trans. Softw. Eng. **32**(9), 627–641 (2006)
3. Arcega, L., Font, J., Haugen, Ø., Cetina, C.: Feature location through the combination of run-time architecture models and information retrieval. In: Grabowski, J., Herbold, S. (eds.) SAM 2016. LNCS, vol. 9959, pp. 180–195. Springer, Cham (2016). doi:10.1007/978-3-319-46613-2_12
4. Basili, V.R.: The role of experimentation in software engineering: past, current, and future. In: Proceedings of the 18th International Conference on Software Engineering, ICSE 1996, pp. 442–449. IEEE Computer Society, Washington, DC (1996)
5. Basili, V.R., Caldiera, G., Rombach, H.D.: The goal question metric approach. In: Encyclopedia of Software Engineering. Wiley (1994)
6. Bencomo, N., Hallsteinsen, S., Santana de Almeida, E.: A view of the dynamic software product line landscape. Computer **45**(10), 36–41 (2012)
7. Cetina, C.: Achieving autonomic computing through the use of variability models at run-time. Ph.D. thesis, Universidad Politécnica de Valencia (2010)
8. Dit, B., Revelle, M., Gethers, M., Poshyvanyk, D.: Feature location in source code: a taxonomy and survey. J. Softw. Maintenance Evol. Res. Pract. **25**(1), 53–95 (2011)
9. Dit, B., Revelle, M., Poshyvanyk, D.: Integrating information retrieval, execution and link analysis algorithms to improve feature location in software. Empirical Softw. Eng. **18**(2), 277–309 (2013)
10. Font, J., Arcega, L., Haugen, Ø., Cetina, C.: Building software product lines from conceptualized model patterns. In: Proceedings of the 2015 19th International Software Product Line Conference, SPLC 2015, Nashville, TN, USA (2015)
11. Font, J., Arcega, L., Haugen, Ø., Cetina, C.: Feature location in model-based software product lines through a genetic algorithm. In: Kapitsaki, G.M., Santana de Almeida, E. (eds.) ICSR 2016. LNCS, vol. 9679, pp. 39–54. Springer, Cham (2016). doi:10.1007/978-3-319-35122-3_3
12. Hulth, A.: Improved automatic keyword extraction given more linguistic knowledge. In: Proceedings of the 2003 Conference on Empirical Methods in Natural Language Processing, EMNLP 2003, Stroudsburg, PA, USA, pp. 216–223. Association for Computational Linguistics (2003)
13. IBM: An architectural blueprint for autonomic computing. Technical report, IBM (2006)
14. Landauer, T.K., Foltz, P.W., Laham, D.: An introduction to latent semantic analysis. Discourse Process. **25**(2–3), 259–284 (1998)
15. Lehman, M.M., Ramil, J., Kahen, G.: A paradigm for the behavioural modelling of software processes using system dynamics. Technical report, Imperial College of Science, Technology and Medicine, Department of Computing, September 2001

16. Liu, D., Marcus, A., Poshyvanyk, D., Rajlich, V.: Feature location via information retrieval based filtering of a single scenario execution trace. In: Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering, ASE 2007, New York, NY, USA, pp. 234–243. ACM (2007)
17. Marcus, A., Sergeyev, A., Rajlich, V., Maletic, J.: An information retrieval approach to concept location in source code. In: Proceedings of the 11th Working Conference on Reverse Engineering, pp. 214–223, November 2004
18. Martinez, J., Ziadi, T., Bissyandé, T.F., Le Traon, Y.: Bottom-up adoption of software product lines: a generic and extensible approach. In: Proceedings of the 19th International Software Product Line Conference, SPLC 2015, Nashville, TN, USA (2015)
19. Muñoz, J.: Model driven development of pervasive systems. building a software factory. Ph.D. thesis, Universidad Politécnica de Valencia (2008)
20. Poshyvanyk, D., Gueheneuc, Y.-G., Marcus, A., Antoniol, G., Rajlich, V.: Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. IEEE Trans. Softw. Eng. **33**(6), 420–432 (2007)
21. Revelle, M., Dit, B., Poshyvanyk, D.: Using data fusion and web mining to support feature location in software. In: IEEE 18th International Conference on Program Comprehension (ICPC), pp. 14–23, June 2010
22. Salman, H.E., Seriai, A., Dony, C.: Feature location in a collection of product variants: combining information retrieval and hierarchical clustering. In: The 26th International Conference on Software Engineering and Knowledge Engineering, Hyatt Regency, Vancouver, BC, Canada, 1–3 July 2013, pp. 426–430 (2014)
23. Salton, G., McGill, M.J.: Introduction to Modern Information Retrieval. McGraw-Hill Inc, New York (1986)
24. Travassos, M.O.B.G.H.: Contributions of in virtuo and in silico experiments for the future of empirical studies in software engineering. In: Proceedings of the Workshop on Empirical Studies in Software Engineering (ESEIW). IEEE Computer Society (2003)

12.5 SANER'16 Paper

Title:	Achieving Knowledge Evolution in Dynamic Software Product Lines.
Authors:	Lorena Arcega, Jaime Font, Øystein Haugen, Carlos Cetina.
Proceedings:	Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, (SANER '16).
Location:	Suita, Osaka, Japan - March 14-18, 2016
Publisher:	IEEE Computer Society
Pages:	505-516
DOI:	`https://doi.org/10.1109/SANER.2016.24`
Contribution:	Lorena Arcega is the main author of the paper and is responsible for 90% of the work. She was also responsible for the oral presentation of the work which took place during the conference.

2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering

Achieving Knowledge Evolution in Dynamic Software Product Lines

Lorena Arcega*†, Jaime Font*†, Øystein Haugen‡, Carlos Cetina*
*San Jorge University, SVIT Research Group, Zaragoza, Spain
Email: {larcega,jfont,ccetina}@usj.es
†University of Oslo, Department of Informatics, Oslo, Norway
‡Østfold University College, Faculty of Computer Science, Halden, Norway
Email: oystein.haugen@hiof.no

***Abstract*—Dynamic Software Product Lines (DSPLs) offer a strategy to deal with software changes that need to be handled at run-time. In response to context changes, a DSPL capitalize on knowledge about the architecture variability of the software system to shift between configurations. Similar to any other kind of software, a DSPL needs to evolve over time but current approaches require software engineers to manually perform the DSPL evolution. Our work addresses the evolution of the architecture variability that makes up the knowledge of the DSPL. Given a new version of the architecture variability, we calculate its configuration space and propose strategies that allow migration from the current version to the new version. Our strategy solves the collision of the realization layer resulting from the integration of the new version of the variability specification. We evaluate our dynamic evolution strategy using the Goal-Question-Metric method for a Smart Hotel case study with 2^{39} possible configurations as starting point. Our experiment indicates that the proposed technique would enable automatic evolution in 9 out of 10 cases. In the rest of the cases, all of the DSPL configurations changed between the old and the new version, which frustrates an automatic evolution.**

I. Introduction

Dynamic Software Product Lines (DSPLs) offer a strategy to deal with software changes that need to be handled at run-time. Specifically, DSPLs shift between different configurations triggered by context changes and are driven by means of the architecture variability knowledge. A recent survey [1] reveals that normally the knowledge of a DSPL is formalized by a variability model and an architecture model described in a Domain Specific Language (DSL). The infrastructure that uses this knowledge for the run-time reconfigurations is a MAPE-K loop [2]. DSPLs exist in several domains such as transportation system's production and warehousing environments [3], recommendation systems [4], autonomous navigation in robots [5], environmental monitoring [6], automotive systems [7] and smart homes [8], [9], [10].

Nevertheless, similar to any other kind of software, a DSPL needs to evolve over time. However, the DSPL evolution has some specific characteristics: (1) the co-evolution of the variability model and the architecture model (if the variability model evolves the architecture model must also evolves and vice versa), and (2) the running system has to be available for the interaction with the context throughout the evolution.

Current research efforts in DSPLs propose the evolution of DSPLs that are already deployed by developing and deploying new software bundles that represent alternative implementations [11], [1], [12], [13]. The integration of new bundles into the DSPL has to be performed manually by software engineers. They have to manually inspect and manipulate the specification of variability and architecture and do not guarantee that the system remains available throughout the evolution of the knowledge of the DSPL.

Our work addresses the knowledge evolution of DSPLs. We propose to address the DSPL knowledge evolution by means of the configuration space of a DSPL. Given a new version of the models of a DSPL, we calculate its configuration space and propose strategies that allow migration from the current version to the new version taking into account shared configurations between their configuration spaces.

One problem that can arise during the evolution is the collision between model elements. A collision is when a feature of the DSPL is realized differently in the current version of the DSPL compared with the new version of the DSPL. Our strategy solves the collision of the realization layer resulting from the integration of the variability specification by taking into account each type of collision depending on its nature. We develop a set of rules for solving each type of collision.

We evaluate our DSPL knowledge evolution strategy using the Goal-Question-Metric method through a simulated evolution of a Smart Hotel DSPL. The Smart Hotel presents thirty-nine features in the feature model, and thirteen services, twenty devices and thirty-five channels in the architecture model. That is, the configuration space of the Smart Hotel has 2^{39} potential configurations. The results of our work demonstrate that the proposed strategies complement the current implementations by solving the automatic evolution of the variability knowledge at run-time in 9 out of 10 cases. In 1 out of 10 cases, our strategy cannot evolve the DSPL automatically because the input models differ greatly from each other, that is, they have no common configurations.

The remainder of the paper is structured as follows. In Section 2, we present the models and the implementation of

978-1-5090-1855-0/16 $31.00 © 2016 IEEE
DOI 10.1109/SANER.2016.24

IEEE computer society

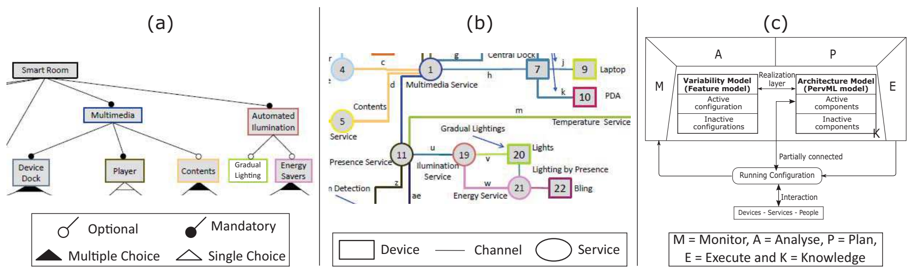


Figure 1. (a) Subset of the Smart Hotel Feature Model, (b) Subset of the PervML model, and (c) MoRE.

the MAPEK loop. In Section 3, we introduce our evolution strategy and the different evolution scenarios. In Section 4, we explain the implementation details of our strategy. In Section 5, we present our evaluation for the Smart Hotel DSPL. In Section 6, we examine the related work, and we present the conclusions in Section 7.

II. Background

DSPLs move existing Software Product Line (SPL) engineering processes to run-time, ensuring that each reconfiguration of the system reach a valid configuration state [1]. Therefore, a DSPL generates a single product which is able to adapt its behaviour at run-time.

Variability modelling, which consist in defining the commonalities and variabilities, is the central activity of SPLs and DSPLs. In a DSPL the variability model describes the variations that can be produced at run-time. The variability model refers to the system architectural components. In DSPLs the system architecture supports all possible configurations defined by the variability model.

The evaluation of this approach is performed through a reconfigurable DSPL for a Smart Hotel [14]. The run-time reconfigurations are performed by an implementation of a MAPE-K loop [2] named Model-based Reconfiguration Engine (MoRE) [14]. Recent surveys reveal that MAPE-K is the most common implementation for reconfiguration loops in DSPLs [1], [15]. This will enable other software engineers to apply these evolution ideas to their MAPE-K DSPL.

A. Smart Hotel Variability Modeling

In the Smart Hotel DSPL, the variability model is expressed through a feature model. The architecture model is defined using a DSL. The realization layer defines the connection between the variability model and the architecture model [16], [17]. Finally, the output system is obtained through a model (DSL) to text (Java code) transformation.

Feature models describe the common and variable characteristics of a system [18]. In feature models, features are hierarchically linked in a tree-like structure through variability relationships (optional, mandatory, or single choice), and are optionally connected by cross-tree constraints (requires or excludes). Our feature model represents all of the different features that the Smart Hotel has implemented.

A feature model contains the set of all features (selected or unselected). A Running Configuration (RC) of a system is defined as the set of all selected features in its feature model at a given time. The subset of the Feature Model in Figure 1 shows a small part of the entire Smart Hotel. The grey features represent the running features of the Smart Hotel, while the white features represent potential variants since they may be activated in the future. For instance, the system can potentially be upgraded with a Gradual lighting.

Although a feature model can represent commonalities and variabilities in a very concise taxonomic form, features in a feature model are merely placeholders. We use a weaving model as the realization layer, that is, for mapping features to architecture model elements. The weaving model expresses a link between a feature model and model elements. This weaving approach enables us to configure architecture models from a set of given features.

We use Pervasive Modelling Language (PervML) [19] to describe the Smart Hotel. PervML is a DSL that describes pervasive systems using high-level abstraction concepts based on Meta-Object Facility (MOF) [20]. However, other MOF-based DSLs for other domains can be used equally well with our approach.

Due to space constraints, in this work, we only focus on the subset of PervML that specifies the relationships among devices and services. This subset specifies the components that define a particular system (services and devices) and how these components are connected with each other (channels). Services are depicted by circles, devices are depicted by squares, and the channels connecting services and devices are depicted by lines (see Figure 1 PervML Model).

B. DSPL Reconfiguration Loop: MoRE (Model-based Reconfiguration Engine)

Control loops have been identified as essential elements to realize the adaptation of software systems. IBM suggested a reference model for autonomic control loops [2], which is

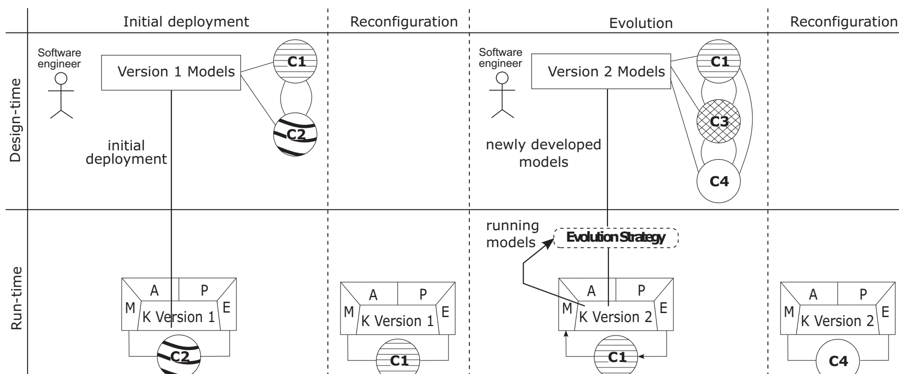


Figure 2. Operations of the DSPL.

called MAPE-K loop. This loop is very useful to researchers that work on run-time variability to make their systems autonomous. The adaptation is based on models (the K element), which means that models are present at run-time [1]. To enable autonomic behaviour, the system must change from one configuration to another by itself. These changes are then translated into reconfiguration actions that modify the system components accordingly.

We use an implementation of the MAPE-K loop called Model-based Reconfiguration Engine (MoRE) [14]. MoRE translates context changes into changes in the activation/deactivation of features. These changes are then translated into the reconfiguration actions that modify the system components accordingly.

In the previous section, we have presented the variability model and the architecture model that MoRE uses as Knowledge (K) to switch between configurations (see Figure 1 (c)). That is, the Smart Hotel DSPL knowledge is composed by the feature model and the PervML architecture model. In MoRE, the Monitor (M) uses the run-time state as input to check context conditions. If any of these conditions are fulfilled, the Analyzer (A) uses the associated resolution and the previous model operations to query the run-time models about the necessary modifications. The response of the models is used by the Planner (P) to elaborate a reconfiguration plan. This plan contains reconfiguration actions, which modify the system architecture and maintain the consistency between the models and the architecture. The Execution (E) of this plan modifies the architecture in order to activate/deactivate the features specified in the resolution.

The reconfiguration of the system is performed by executing reconfiguration actions that deal with the activation and deactivation of components and the creation and destruction of channels among components. For example, the Java virtual machine allows loading and unloading code dynamically and component platforms allow loading, connecting and disconnecting component instances.

The feature model specifies the possible configurations of the system, while the PervML architecture model can be rapidly retargeted to a specific configuration in response to changes in the context. MoRE calculates the architecture increments and decrements in order to determine the actions necessary to modify the system architecture. These operations take a feature resolution as input, and they calculate the modifications to the architecture in terms of devices, services, and channels.

Moreover, it is absolutely necessary to have a way to analyze the reconfigurations before performing them. MoRE validates the configurations resulting from the simultaneous fulfillment of context events at design-time. Therefore, unexpected configurations can be avoided. Specifically, MoRE analyzes variability models by means of the FAMA framework [21] for variability analysis. FAMA framework integrates some of the most commonly used logic representations and solvers proposed in the literature. This framework enables to determine if a system configuration is valid, and it can also provide explanations about invalid configurations.

III. Our Evolution Strategy

Figure 2 shows the operation of the DSPL presented. The upper part depicts the actions performed at design-time by software engineers while the lower part shows the impact that those actions have on the running DSPL.

The first column presents the initial deployment of the DSPL; the first versions of the models are created by the software engineer (the upper part). The initial deployment of the DSPL is performed using those models. The lower part of the first column shows how the DSPL is conformed

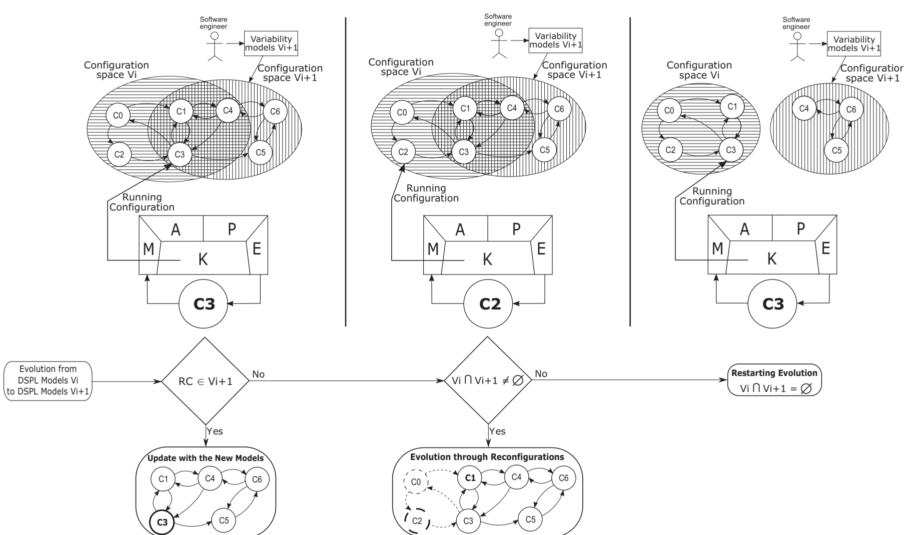


Figure 3. Different Evolution Scenarios.

with the MAPE-K loop. The DSPL can be reconfigured at run-time by hot-swapping existing components.

In the second column of Figure 2, the system is reconfigured and starts using a configuration described by the variability knowledge that is different from the original one (in this case, a shift from Configuration 2 (C2) towards Configuration 1 (C1)). This reconfiguration is performed at run-time; an event from the physical world triggers the use of a new configuration. For example, in the Smart Hotel, C1 is the configuration when a room is empty (i.e., the sensor of the room are used for security purposes). When the client of the hotel enters in the room, MoRE performs the reconfiguration and changes from C1 to C2 that is when the client is in the room (i.e., the sensors of the room are used for illumination purposes).

The third column presents an example of evolution of the DSPL. The software engineer creates the new models (the top part of third column). Our strategy is used to translate those changes to the running DSPL. The current DSPL (Version 1) will be evolved with the new DSPL (Version 2) as indicated by the strategy in order to modify the running DSPL without suspending its execution. That is, some of the configurations are added and some others are removed without stopping the system. When a configuration is modified, we assume that the configuration has been removed and a new configuration has been added.

After the evolution, the system can perform new reconfigurations. In the fourth column of Figure 2, the system is reconfigured from Configuration 1 (C1) to Configuration 4 (C4). The system is able to reach configurations of the new version of the models.

This work focuses on the knowledge evolution of a DSPL. We consider evolution to be the integration of newly developed components without having to stop the system.

We show the evolution by means of the configuration space that is defined by the models. A configuration space is composed of a set of configurations and transitions between configurations. Each of the configurations is composed of a set of active features defined in the variability model. In turn, an active feature is related to a set of architecture model elements by means of a realization layer. The system source code is obtained through a model to text transformation taking as input the complete architecture model.

Our evolution starts with a design-time evolution when the software engineer develops a new configuration space at design-time. In our case, the design-time evolution is the enabler for the DSPL knowledge evolution [22].

Once the evolution at design-time is developed by the software engineer, our strategy allows the run-time activation of the new configurations without having to stop the system. Our strategy distinguishes between three main scenarios.

A. Evolution scenarios

Figure 3 depicts the different evolution scenarios. The software engineer develops a new variability model which defines a new configuration space. We distinguish three

scenarios taking into account the current configuration space, the newly developed configuration space, and the running configuration.

In the first case, there is an intersection between the two configuration spaces, that is, some configurations belong to both. In addition, in the first case, the running configuration belongs to this intersection (i.e., the running configuration belongs to both configuration spaces). Hence, the running configuration is also in the new configuration space.

When this case occurs, our strategy performs an **Update**. An update requires the evolution of the model elements that are not involved in the running configuration. Our strategy removes the old configurations and adds the new ones.

The first column in Figure 3 shows an update when the running configuration is C3. Configuration C3 is in version Vi models and in version Vi+1 models; hence, C3 is in the new version. The resultant configuration space only contains the configurations of version Vi+1 (C1, C3, C4, C5, and C6).

In the second case, there is also an intersection between the two configuration spaces. However, the running configuration does not belong to this intersection, that is, the running configuration is not part of the new version of the configuration space. The running configuration is only in the old version of the configuration space.

In this scenario, our strategy performs an **Evolution through Reconfigurations**. Our strategy composes the old configuration space with the new configuration space. The old version is used in a transient period until a configuration of the new version is reached. In other words, the two versions coexist until the running configuration reaches a configuration of the new version, which allows the safe removal of the old version of the configuration space.

The second column in Figure 3 shows the evolution when the running configuration is C2. Configuration C2 is not present in version Vi+1 models. The resultant configuration space is a composition of the old configurations (C0, C1, C2, C3, and C4) with the new configurations (C1, C3, C4, C5, and C6). This is only a transient situation; the final configuration space only maintain the new configurations (C1, C3, C4, C5, and C6).

The third column in Figure 3 shows that there is no intersection between the two configuration spaces. Hence, the strategy cannot reach the new configuration space; there are no transitions between old configurations and new configurations. Therefore, our strategy is not able to perform an automatic evolution without stopping the system. This scenario needs a **Restarting Evolution**.

B. The special case of a bug

This evolution also allows some configurations to be blocked. For example, if a serious bug has been spotted in the running version and some of its configurations should be avoided (also in the transient situation), the software engineer can develop a new configuration space without the configuration that contains the bug.

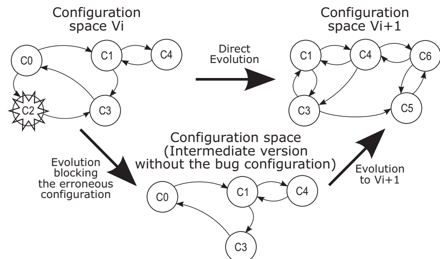


Figure 4. Example of the Special Case of a Bug.

In the same way as in the evolution scenarios, if the erroneous configuration is not running, our strategy simply performs an **Update** discarding the erroneous configuration. However, if the erroneous configuration is running, our strategy performs an **Evolution through Reconfigurations**. Once the system leaves the erroneous configuration, the system cannot return to it. This is how the system eliminates the bug.

In the example of Figure 4, the software engineer wants to evolve the DSPL from version Vi models to version Vi+1 models. However, there is a bug in configuration C2. Instead of performing the evolution directly from version Vi to version Vi+1, the best way to block the erroneous configuration, C2, is to develop an intermediate version of the configuration space that does not contain this erroneous configuration. Then, the first evolution is from version Vi to the intermediate version. Our strategy applies one kind of evolution or the other depending on the configuration that is running.

Once this intermediate evolution is performed, the configuration that contains the bug has been removed. Thus, the system is prevented from repeatedly reaching the erroneous configuration. Finally, another evolution from the intermediate version to the version Vi+1 is needed to achieve the desired configuration space.

IV. Model Operations to Realize the Evolution Strategy

This section introduces the model operations of our evolution strategy. The evolution starts when the software engineer develops the new version of the models at design-time. These new models define the new configuration space. Then, our strategy evolves the configuration space of the Smart Hotel DSPL. The strategy takes as input the new version of the models developed at run-time and the running models of the Smart Hotel.

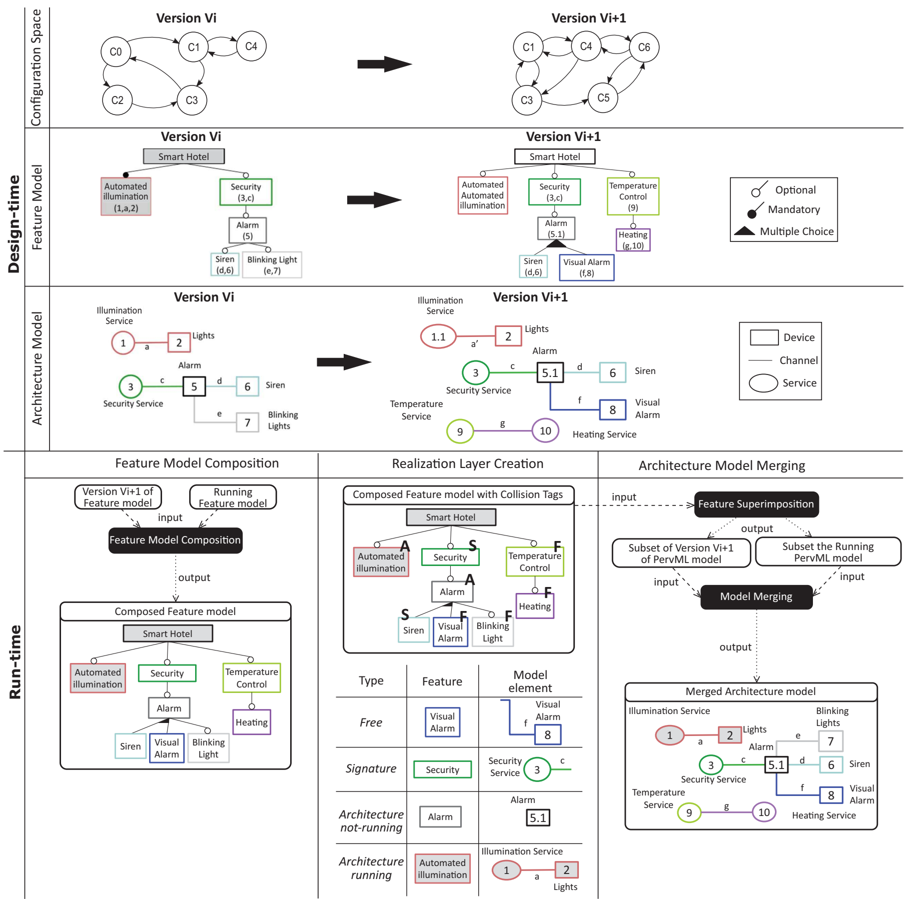


Figure 5. Evolution of the Smart Hotel

A. Design-time Evolution

This section shows the evolution of a Smart Hotel DSPL example performed by the software engineer at design-time. The upper part of Figure 5 depicts the model versions and the evolutions performed at design-time. Figure 5 shows the configuration space, the DSPL Feature Model, and the DSPL PervML Model. The information in brackets after the name of each element in the feature model corresponds to the links to the architecture model.

The upper part of Figure 5 shows the changes performed by the software engineer at design-time over the configuration space, the variability model, and the architecture model. Version Vi shows the first version of the Smart Hotel models. In this small example, the Smart Hotel has two services, automated illumination and security services. The automated illumination feature enables lights to be automatically controlled. The security service is used to alert people of critical notifications such as fire or water leaks in the room. The security service can have a traditional audible alarm or blinking lights.

Version Vi+1 of Figure 5 shows the new models created by software engineers. They create the temperature control service, which takes into account the occupancy of the room and the actual temperature in order to adjust the heating system. They also change the blinking light alarm

feature to the visual alarm feature. If any critical situation occurs, the visual alarm shows a message with instructions; therefore, the visual alarm shows necessary information that the blinking lights do not show.

B. DSPL Knowledge Evolution at Run-time

Our strategy distinguishes three scenarios to perform the evolution. In the **Update** evolution, the running models are changed for the new ones. An update requires the evolution of the model elements that are not involved in the running configuration. Our strategy removes the old configurations and adds the new version of the configuration space.

In the **Evolution through Reconfigurations** scenario, our strategy has a transient situation before reaching the final configuration space. The transient situation is formed by a composition of the new models developed at design-time and the running models of the DSPL.

To get the transient situation, our strategy composes the new version of the feature model and the running feature model, and merges the new version of the architecture model and the running architecture model. A new realization layer is created to map each feature with the architecture elements that are related to it. Our strategy implements this composition of the models as indicated below.

1) Feature Model Composition: Our strategy combines the running feature model and the new version of the feature model to produce the new version of the running feature model. This composed feature model specifies configurations that may be reached after the DSPL knowledge evolution. To do the variability composition, our strategy extends the *Reference-based and Slicing* composition process implemented in [23].

The composition process consists of two main phases: (1) the matching phase identifies model elements that describe the same concepts in the input models; (2) the merging phase groups the matched elements together to create new elements in the resulting model. We chose the hybrid implementation, *Reference-Based and Slicing* presented in [23], because it is the most customizable implementation in their study.

The key idea of *Reference-Based* implementation is to build a separate feature model that contains features with the same names as the input feature models. The features are then related to input features through a set of logical constraints. We use *Slicing* to eliminate internal variables, which are needed to perform the composition (because they increase the computational time). The result is a feature model that joins the input variability models and the constraints.

Since conflicts between constraints from different versions of the feature model can occur, our strategy is driven by configuration semantics (the architecture configuration set of the composed variability model is the union of the architecture configuration sets of the input variability models). However, our current implementation of the strategy does not take into account ontological semantics to determine the most appropriate variability hierarchy. Given a set of configurations, there still exist different candidate variability models with other different hierarchies [18].

The Feature Model Composition in Figure 5 depicts an example of the feature model composition of our Smart Hotel example. The inputs of the composition are the running feature model and the version Vi+1 of the feature model developed by the software engineer. The output of the operation is the composed feature model (see Figure 5 lower part).

2) Realization Layer Creation: In order to link the composed variability model to the architecture model, the strategy generates a new realization layer. The realization layer creates a link between features and architecture model elements. In the feature model composition, some collisions between features can occur if the features are not unique. In our case, a feature is not unique when in the composition there is another feature with the same signature. Our strategy only considers the name property of a feature model element as signature. In other words, two features match if they have the same name property.

We follow a set of rules to create the realization layer depending on the collisions among features detected during the matching phase of the feature model composition. A collision depends on the uniqueness of the features. To distinguish the type of collision, we extend our implementation of the feature model composition to add a tag to each feature.

- A feature of type *free (F)* means no collision, the feature is only found in one of the feature models.
- A feature of type *signature (S)* means a collision of the features, where their references to the architecture model are the same. In other words, there are two features (one in each feature model) with the same signature, where the model architecture elements related to them are the same for both features.
- A feature of type *architecture (A)* means a collision of the features and of the architecture model elements related to them (i.e., the architecture elements related to these features are different).

The composed feature model in the realization layer creation column of Figure 5 shows the features and the type of each feature.

The creation of the realization layer follows one rule with each type of feature. For features of type *free (F)* and *signature (S)*, our strategy holds the current architecture elements related to them. This is because there are no collisions in the architecture elements related to the features.

For features of type *architecture (A)*, we differentiate two cases. For active features (involved in the running configuration), our strategy keeps the architecture elements denoted by the run-time variability model before the DSPL knowledge evolution. For inactive features (those not involved in the running configuration), our strategy updates

the architecture elements with the new ones that come from the new variability model of the design-time.

The realization layer creation column of Figure 5 shows the different types of features and the model elements associated with each one after creating the realization layer. For example, the illumination service and the alarm have changed in the new version of the architecture model (see Figure 5, upper part). However, since the illumination service is active in the running configuration, the architecture model related to this feature remains unchanged. In contrast, the alarm is changed to the new version because it does not belong to the running configuration; it is inactive.

3) Architecture model merging: Finally, our strategy merges the architecture models that are driven by the realization layer. It performs the superimposition operation ($\odot$) [14] over the variability model. The superimposition takes a feature and returns the set of architecture model elements that are related to this feature. The result is a subset of the running architecture model and another subset of the new version of the architecture model. These two subsets are merged to create the new version of the running architecture model. Our strategy uses a signature-based model composition [24] to achieve the merging.

The model merging is structured in two phases, the matching phase and the merging phase. In the matching phase, model elements that are described in different models are identified. In the merging phase, matched model elements are merged to create a new model element.

To support automated element matching, each element is associated with a signature type that determines the uniqueness of each element. Two elements with equivalent signatures cannot coexist in a model. Our strategy only considers the name property of a model element as signature. In other words, two elements match if they have the same name property.

The merged model contains the union of the model elements in the source models; matching elements are included only once in the merged model. The new version of the running architecture model in Figure 5 shows an example of a merge model.

The superimposition operator ($\odot$) [14] is used to query a realization layer to identify which architecture model elements support a certain feature. The superimposition takes a feature and returns the set of components and channels that are related to this feature. By means of the superimposition operation, it is possible to project a particular feature to the architecture components.

Our strategy performs the superimposition operation taking the composed feature model. This operation returns different model elements to support all possible feature model configurations. These elements are matched and merged as a result of the composed model.

The resultant models are only a transition. Once our reconfiguration engine reaches a new configuration, our strategy removes the old elements. The final models only contain elements of the new version. Therefore, the system can only reach configurations belonging to the new version.

V. Validation of the DSPL knowledge evolution of the Smart Hotel

In this section we aim to show the applicability of our evolution strategy, by showing that it can support in practice the evolution of a Smart Hotel DSPL. We do not focus on computational complexity or scalability because these are properties that emerge from the choice of expressive power of the language used to express the variability and the choice of the reconfiguration loop implementation.

We conducted a validation of our strategy using a Smart Hotel case study. The Smart Hotel used in the previous sections to explain our strategy is a subset of the real Smart Hotel that we used in this validation. This Smart Hotel was developed previously [14]. The Smart Hotel reconfigures its services according to changes in the surrounding context. A hotel room changes its features depending on users activities to make their stay as pleasant as possible.

According to the feature model, the Smart Hotel presents thirty-nine features and six cross-tree constraints. The main concepts of the Smart Hotel DSPL architecture are services, devices, and the communication channels among them. The Smart Hotel has thirteen services, twenty devices and thirty-five channels. That is, the configuration space of the Smart Hotel has 2^{39} possible configurations.

We defined the experimental design of our study using the Goal-Question-Metric method (GQM) [25] and its template [26]. The GQM method was defined as a mechanism for defining and interpreting a set of operation goals using measurements. In this experiment, our goal was the following:

- Object : Our Smart Hotel DSPL
- Purpose: Validation
- Issue: The applicability of the automatic evolution strategy
- Context: Evolution of the DSPL architecture variability knowledge

To fulfill this goal, we focused on answering this research question: Do the scenarios of our strategy cover the evolution of the Smart Hotel DSPLs?

Basili [25] and Travassos [27] describe four kinds of studies: in-vivo, in-vitro, in-virtuo, and in-silico. In our case, we chose to carry out in-silico experiments, where subjects and the real world are described as computer models. The environment was composed entirely of computer models, with which human interaction was reduced to a minimum. This offers major advantages regarding the cost and the feasibility of replicating a real-world configuration. In addition, some scenarios such as fires or floods cannot be replicated in the real world.

Moreover, we simulated the evolution by means of a simulation approach for exploring the effects of product line

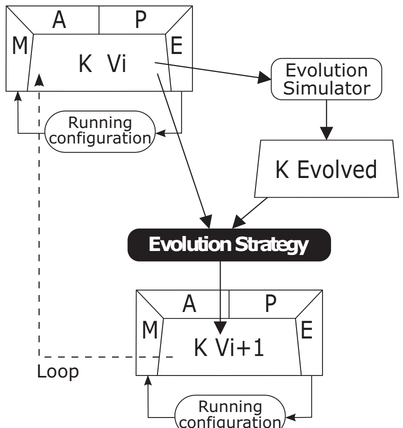


Figure 6. Evaluation Steps.

evolution [28]. The simulator tool allows generating random models based on product line profiles.

The evolution profiles are *Product line refactoring* that leads to the adaptation of existing components or the development of new ones, *Product placement* that changes the problem space to allow different configurations of the existing components, without changing them, and *Continuous evolution* that is subject to continuous changes in both spaces by adding/removing components or reorganizing decisions.

We implemented the above evolution profiles in a Evolution Simulator. This Evolution Simulator is based on Ecore-mutator [29], which is an EMF-based framework to mutate EMF models that conform to an Ecore metamodel. In particular, we implemented custom mutators that, given a model as input, add, remove or modify model elements according to each evolution profile. Our Evolution Simulator randomly chooses which evolution profile of the ones presented in this section is going to be applied.

Figure 6 shows the steps followed in this evaluation. The Evolution Simulator takes one version of the models as input and returns an evolved version of these models. Next, our evolution strategy performs the DSPL knowledge evolution. The input for the strategy was the running models and the new version of the evolved models developed in the Evolution Simulator. The output of the strategy is an evolved DSPL knowledge placed on the reconfigurable system. We performed simulations until all evolution profiles were covered.

For example, in the first evolution, the Evolution Simulator took as input the first version of the Smart Hotel DSPL (2^{39} configurations). The new version of these models (output of Evolution Simulator) had forty-two features in the feature model, fifteen services, twenty-four devices, and forty-five channels in the architecture model and 2^{42} potential configurations in the configuration space. This evolution profile corresponded to a *Continuous evolution* because five new configurations were added, two old configurations were removed and six of the old ones were modified.

Next, our evolution strategy evolved the models from the Smart Hotel DSPL. Our strategy took as input the new version of the models. As the running configuration was not in the configuration space of the new version of the models, the evolution was performed by an **Evolution through Reconfigurations**. Finally, the Smart Hotel DSPL could reach 2^{42} different configurations with the new version of the models instead of the 2^{39} configurations that the Smart Hotel DSPL could reach with the previous models.

A. Scenarios covered by the strategy

After analysing the data obtained from the evolutions, we obtain that in 1 out of 4 cases of the evolutions, the running configuration remains in the configuration space of the new version of the models. The strategy can perform a direct **Update** scenario without interrupting the running configuration.

In 2 out of 3 cases of the evolutions, the running configuration does not belong to the configuration space of the new version of the models. The strategy is able to find a way from the running configuration to reach a configuration that belongs to the configuration space of the new version of the models. Hence, the strategy performs an **Evolution through Reconfigurations** scenario.

However, in 1 out of 10 cases of the evolutions, the strategy cannot find a way to reach a configuration that belongs to the configuration space of the new version of the models. The system cannot be evolved with our automatic strategy and must be stopped to perform the evolution.

We have observed that the models corresponding to this 1 out of 10 cases that cannot be evolved automatically are models that differ greatly from each other. The **Restarting** scenario is required when none of the old configurations are retained in the new version of the configuration space. In a no simulated environment, the evolution of a system in which all of the configurations change without keeping at least one old configuration in the new version is very uncommon. Although it has occurred in an exhaustive simulation, it means not having in common any of those 2^{39} configurations between the old and the new version.

B. Threats to Validity

Our validation exhibits some applicability. Regarding the generalization, our results and findings are based on a single DSPL in the domain of the Smart Hotels. However, given the scale and complexity of our Smart Hotel (thirty-nine features, thirteen services, twenty devices, and thirty-five

channels in the PervML model and 2^{39} possible configurations in the configuration space), we consider our validation a good starting point representing a realistic case.

Furthermore, the election of the Smart Hotel DSPL for the validation was motivated because the Smart Hotel adheres to the the core ideas of DSPLs such as MAPE-K loop and architecture variability knowledge [1]. Although, the evolution strategy was validated by means of the Smart hotel, the ideas of the evolution strategy are general and they can be applied to other DSLPs which are base on MAPE-K loop and architecture variability knowledge.

Concerning the evaluation runs, our validation may not seem sufficient for a continuously operating system. However, each run covers one complete continuous life-cycle of the system.

Finally, to ensure the validity of the case study, the validation was done by a student in his last year of his master as part of his master's thesis. The participation of the authors of the strategy was limited to explaining the operation of the strategy so that the student could correctly perform the validation. This student was responsible for developing the Evolution Simulator and for conducting the validation with the Smart Hotel case study. Thus, we have achieved that the validation was independent of the main research.

VI. Related Work

Hallsteinsen et al. [11] develop the MUSIC framework. MUSIC supports the dynamic addition and removal of components and service variants, as well as compositions with their own set of model fragments that describe internal variability. Their evolution is focused on software by means of developing new software bundles. However, their evolution must be performed manually by a software engineer, while our evolution strategy allows an automatic evolution.

Pascual et al. [30] present an approach that provides support for the dynamic reconfiguration of mobile applications, optimizing their architectural configuration according to the available resources. They model the variability of the application's software architecture using Common Variability Language (CVL) [16] while we use a feature model and a Domain Specific Language (DSL) to express the variability. They evolution is performed thought a genetic algorithm whilst we develop different operation (i.e., feature model composition or model merging) for each one of the models.

Perrouin et al. [31] use a MAPE loop to manage another MAPE loop. Their approach not only adapts the system, it also adapts both its adaptation mechanism and its adaptation policies. They define a dynamically reconfigurable adaptation loop. The dynamic reconfiguration of this adaptation loop is achieved by employing adaptation techniques that are similar to the ones used to adapt the system itself. They focus on evolving adaptation rules or reconfiguration scripts, while we focus our work on DSPL knowledge evolution that incorporates knowledge from new versions of design-time models into run-time models. The combination of the two approaches may improve current DSPL implementations.

Hussein et al. [32] develop an approach to enable the run-time evolution of context-aware adaptive services. Their approach captures a service's model from three aspects: functionality, context, and adaptive behaviour. Thus, these aspects and their relationships can be captured and manipulated at run-time. With this approach, the software engineer can perform the service's run-time changes at the modelling level. However, they compute the differences between the evolved model and the initial model and generate adaptation actions. Then, this actions are applied to the service's run-time artifacts. We take into account the run-time model and the run-time configuration to apply our strategy, hence we only need to perform adaptation action when the running configuration does not allow the evolution.

Capilla et al. [15] provides an overview of the state of the art and current techniques that face the challenges of run-time variability in the context of Dynamic Software Product Lines. They propose a solution for the automation of changes in the structural variability (i.e., adding or removing features at run-time). They use the notion of super-types [33], [34] while we use the feature model composition. In addition, they propose some techniques to check the feature model and the configuration model [21], [35], [36] once they are modified. These techniques could be applied to our strategy in the future to ensure that the evolved models are valid.

VII. Conclusions

This work addresses the knowledge evolution of DSPLs by means of the configuration space of a DSPL. Specifically, our evolution strategy distinguishes three main scenarios taking into account the running configuration. In addition, our strategy solves the collision between components resulting from the evolution. Finally, the system evolves automatically thus enabling the DSPL to reach new configurations.

The proposed automatic evolution strategy is not restricted to the case study we have chosen to evaluate, it can also be applied to a wide range of DSPL domains. The ideas of the evolution strategy can be applied to the most common infrastructure of DSLPs [1], which combines MAPE-K loop and architecture variability knowledge.

In this paper, we have validated the benefits of performing the evolution automatically. The validation of our strategy in the Smart Hotel DSPL has shown promising results in 9 out of 10 cases, which confirm the potential of applying our automatic evolution strategy.

In the near future, we would like to explore automatic evolution in other kinds of systems with other types of variability management, such as systems that use the Common Variability Language (CVL) [16]. We plan to investigate an automatic evolution that covers the maximum number of possible cases.

References

[1] N. Bencomo, S. O. Hallsteinsen, and E. S. de Almeida, "A view of the dynamic software product line landscape," *IEEE Computer*, vol. 45, no. 10, pp. 36–41, Oct 2012.

[2] IBM, "An architectural blueprint for autonomic computing," IBM, Tech. Rep., 2006.

[3] A. Helleboogh, D. Weyns, K. Schmid, T. Holvoet, K. Schelfthout, and W. V. Betsbrugge, "Adding variants on-the-fly: Modeling meta-variability in dynamic software product lines," in *Proceedings of the 3rd International Workshop on Dynamic Software Product Lines (DSPL '09)*, San Francisco, California, USA, Aug 2009, pp. 19–27.

[4] G. H. Alfrez and V. Pelechano, "Context-aware autonomous web services in software product lines," in *Proceedings of the 15th International Software Product Line Conference (SPLC '11)*, Munich, Germany, Aug 2011, pp. 100–109.

[5] M. Kim, J. Kim, and S. Park, "Tool support for quality evaluation and feature selection to achieve dynamic quality requirements change in product lines," in *Proceedings of the 2nd International Workshop on Dynamic Software Product Lines (DSPL '08)*, Limerick, Ireland, Sep 2008, pp. 69–78.

[6] S. Hallsteinsen, S. Jiang, and R. Sanders, "Dynamic software product lines in service oriented computing," in *Proceedings of the 3rd International Workshop on Dynamic Software Product Lines (DSPL '09)*, San Francisco, California, USA, Aug 2009, pp. 28–34.

[7] H. Shokry and M. A. Babar, "Dynamic software product line architectures using service-based computing for automotive systems," in *Proceedings of the 2nd International Workshop on Dynamic Software Product Lines (DSPL '08)*, Limerick, Ireland, Sep 2008, pp. 53–58.

[8] R. Ali, R. Chitchyan, and P. Giorgini, "Context for goal-level product line derivation," in *Proceedings of the 3rd International Workshop on Dynamic Software Product Lines (DSPL '09)*, San Francisco, California, USA, Aug 2009, pp. 8–17.

[9] J. Lee, J. Whittle, and O. Storz, "Bio-inspired mechanisms for coordinating multiple instances of a service feature in dynamic software product lines," *The Journal of Universal Computer Science*, vol. 17, no. 5, pp. 670–683, 2011.

[10] C. Cetina, P. Giner, J. Fons, and V. Pelechano, "Designing and prototyping dynamic software product lines: Techniques and guidelines," in *Proceedings of the 14th International Software Product Line Conference (SPLC '10)*, Jeju Island, South Korea, Sep 2010, pp. 331–345.

[11] S. O. Hallsteinsen, K. Geihs, N. Paspallis, F. Eliassen, G. Horn, J. Lorenzo, A. Mamelli, and G. A. Papadopoulos, "A development framework and methodology for self-adapting applications in ubiquitous computing environments," *Journal of Systems and Software*, vol. 85, no. 12, pp. 2840–2859, Dec 2012.

[12] C. Cetina, P. Giner, J. Fons, and V. Pelechano, "Autonomic computing through reuse of variability models at runtime: The case of smart homes," *IEEE Computer*, vol. 42, no. 10, pp. 37–43, Oct 2009.

[13] M. Hinchey, S. Park, and K. Schmid, "Building dynamic software product lines," *IEEE Computer*, vol. 45, no. 10, pp. 22–26, Oct 2012.

[14] C. Cetina, "Achieving autonomic computing through the use of variability models at run-time," Ph.D. dissertation, Universidad Politcnica de Valencia, 2010.

[15] R. Capilla, J. Bosch, P. Trinidad, A. Ruiz-Cortés, and M. Hinchey, "An overview of dynamic software product line architectures and techniques: Observations from research and industry," *Journal of Systems and Software*, vol. 91, pp. 3–23, May 2014.

[16] Ø. Haugen, B. Mller-Pedersen, J. Oldevik, G. K. Olsen, and A. Svendsen, "Adding standardized variability to domain specific languages," in *Proceedings of the 12th International Software Product Line Conference (SPLC '08)*, Limerick, Ireland, Sep 2008, pp. 139–148.

[17] A. Svendsen, X. Zhang, R. Lind-Tviberg, F. Fleurey, y. Haugen, B. Mller-Pedersen, and G. K. Olsen, "Developing a software product line for train control: A case study of cvl," in *Proceedings of the 14th International Conference on Software Product Lines (SPLC '10)*, Jeju Island, South Korea, Sep 2010, pp. 106–120.

[18] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki, "Reverse engineering feature models," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*, Waikiki, Honolulu , HI, USA, May 2011, pp. 461–470.

[19] J. Muoz, "Model driven development of pervasive systems. building a software factory," Ph.D. dissertation, Universidad Politcnica de Valencia, 2008.

[20] "Meta object facility (mof), 2.0 core specification," 2003, version 2.

[21] D. Benavides, P. Trinidad, and A. Ruiz-Cortés, "Automated reasoning on feature models," in *Proceedings of the 17th International Conference on Advanced Information Systems Engineering (CAiSE '05)*, Porto, Portugal, Jun 2005, pp. 491–503.

[22] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel, "Towards a taxonomy of software change: Research articles," *Journal of Software Maintenance and Evolution: Research and Practice - Unanticipated Software Evolution*, vol. 17, no. 5, pp. 309–332, Sep 2003.

[23] M. Acher, B. Combemale, P. Collet, O. Barais, P. Lahire, and R. France, "Composing your compositions of variability models," in *Model-Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science, A. Moreira, B. Schtz, J. Gray, A. Vallecillo, and P. Clarke, Eds. Springer Berlin Heidelberg, 2013, vol. 8107, pp. 352–369.

[24] R. B. France, F. Fleurey, R. Reddy, B. Baudry, and S. Ghosh, "Providing support for model composition in metamodels," in *Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference (EDOC '07)*, Annapolis, Maryland, USA, Oct 2007, pp. 253–266.

[25] V. R. Basili, "The role of experimentation in software engineering: Past, current, and future," Berlin, Germany, Mar 1996, pp. 442–449.

[26] V. R. Basili, G. Caldiera, and H. D. Rombach, "The goal question metric approach," in *Encyclopedia of Software Engineering*. John Wiley & Sons, 1994.

[27] G. H. Travassos and M. de Oliveira Barros, "Contributions of in virtuo and in silico experiments for the future of empirical studies in software engineering," in *Proceedings of the ESEIW 2003 Workshop on Empirical Studies in Software Engineering (WSESE '03)*, Roman Castles, Italy, Sep 2003.

[28] W. Heider, R. Froschauer, P. Grnbacher, R. Rabiser, and D. Dhungana, "Simulating evolution in model-based product line engineering," *Information and Software Technology*, vol. 52, no. 7, pp. 758–769, Jul 2010.

[29] "Eclipse foundation, ecore-mutator." [Online]. Available: https://code.google.com/a/eclipselabs.org/p/ecore-mutator/

[30] G. G. Pascual, R. E. Lopez-Herrejon, M. Pinto, L. Fuentes, and A. Egyed, "Applying multiobjective evolutionary algorithms to dynamic software product lines for reconfiguring mobile applications," *Journal of Systems and Software*, vol. 103, pp. 392–411, May 2015.

[31] G. Perrouin, B. Morin, F. Chauvel, F. Fleurey, J. Klein, Y. Le Traon, O. Barais, and J. M. Jezequel, "Towards flexible evolution of dynamically adaptive systems," in *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*, Zurich, Switzerland, Jun 2012, pp. 1353–1356.

[32] M. Hussein, J. Han, J. Yu, and A. Colman, "Enabling runtime evolution of context-aware adaptive services," in *Proceedings of the IEEE 10th International Conference on Services Computing (SCC '13)*, Santa Clara, CA, Jun - Jul 2013, pp. 248–255.

[33] O. Ortiz, A. B. García, R. Capilla, J. Bosch, and M. Hinchey, "Runtime variability for dynamic reconfiguration in wireless sensor network product lines," in *Proceedings of the 6th International Workshop on Dynamic Software Product Lines (DSPL '12)*, Salvador, Brazil, Sep 2012, pp. 143–150.

[34] J. Bosch and R. Capilla, "Dynamic variability in software-intensive embedded system families," *IEE Computer*, vol. 45, no. 10, pp. 28–35, Oct 2012.

[35] J. White, D. Benavides, D. C. Schmidt, P. Trinidad, B. Dougherty, and A. Ruiz-Cortes, "Automated diagnosis of feature model configurations," *Journal of Systems and Software*, vol. 83, no. 7, pp. 1094–1107, Jul 2010, {SPLC} 2008.

[36] P. Trinidad, "Automating the analysis of stateful feature models," Ph.D. dissertation, Universidad de Sevilla, 2012.

13

Bug Localization in Models

Contents

13.1 ISD'17 Paper 197
13.2 MODELS'18 Paper 215
13.3 SOSYM'19 Paper 227

13.1 ISD'17 Paper

Title:	On the Influence of Modification Timespan Weightings in the Location of Bugs in Models.
Authors:	Lorena Arcega, Jaime Font, Øystein Haugen, Carlos Cetina.
Proceedings:	Proceedings of the 26th International Conference on Information Systems Development, (ISD '17)
Location:	(University of Central Lancashire Cyprus) Larnaca, Cyprus - September 6-8, 2017
Publisher:	Springer
Pages:	169-185
DOI:	`https://doi.org/10.1007/978-3-319-74817-7_11`
Contribution:	Lorena Arcega is the main author of the paper and is responsible for 90% of the work. She was also responsible for the oral presentation of the work which took place during the conference.

On the Influence of Modification Timespan Weightings in the Location of Bugs in Models

Lorena Arcega, Jaime Font, Øystein Haugen and Carlos Cetina

Abstract Bug location is a common task in Software Engineering, specially when maintaining and evolving software products. When locating bugs in code, results depend greatly on the way code modification timespans are weighted. However, the influence of timespan weightings on bug location in models has not received enough attention yet. Throughout this paper, we analyze the influence of several timespan weightings on bug location in models. These timespan weightings guide an evolutionary algorithm, which returns a ranking of model fragments relevant to the solution of a bug. We evaluated our timespan weightings in BSH, a real-world industrial case study, by measuring the results in terms of recall, precision, and F-measure. Results show that the use of the most recent timespan model modifications provide the best results in our study. We also performed a statistical analysis to provide evidence of the significance of the results.

Keywords Bug location · Model driven engineering · Reverse engineering

A prior version of this paper has been published in the ISD2017 Proceedings (http://aisel.aisnet.org/isd2014/proceedings2017).

L. Arcega (✉) · J. Font · C. Cetina
Universidad San Jorge, Saragossa, Spain
e-mail: larcega@usj.es

J. Font
e-mail: jfont@usj.es

C. Cetina
e-mail: ccetina@usj.es

L. Arcega · J. Font
University of Oslo, Oslo, Norway

Ø. Haugen
Østfold University College, Halden, Norway
e-mail: jfont@usj.es

© Springer International Publishing AG, part of Springer Nature 2018

N. Paspallis et al. (eds.), *Advances in Information Systems Development*,
Lecture Notes in Information Systems and Organisation 26,
https://doi.org/10.1007/978-3-319-74817-7_11

1 Introduction

During software evolution, the existing software of a project undergoes modifications to satisfy changes. A change may result in either the addition of a new software function, the removal of a bug or defect, or the improvement of an existing software functionality. These maintenance and evolution activities take up to 80% of the lifetime of a system [1]. Software maintainers spend from 50% up to almost 90% of their time trying to understand a program in order to make changes correctly [2]. One of the key issues to achieve this goal is finding relevant locations to address the changes.

Bug Location is one of the most important and common activities performed during software maintenance and evolution [3]. Currently, research efforts in Bug Location are concerned with identifying software artifacts associated with bug descriptions. However, most research on Bug Location targets code [4] as the software artifact that realizes the feature, neglecting other software artifacts such as models.

In order to locate bugs in code, the most recent code modifications are regarded as the most relevant. Bug location results depend greatly on the way in which the modification timespans are weighted. The consideration of timespans is based on the Defect Localization Principle. This principle is based on the observation that the most recent modifications to a project are most likely the cause of future bugs [5, 6]. Considering recent project modifications, it is possible to find relevant code for bug location [7].

We perform Bug Location in Models (BLiM). To do so, we locate the most relevant model fragments for a particular bug description. Model fragments are formed by model elements, and each model element has an associated modification time. When we apply the Defect Principle to model fragments, we have to decide how to assign a modification time to the model fragment from the modification time information on its model elements. The contribution of this work is the design, application for BLiM, and evaluation of four fitness functions regarding modification timespan weightings. The weightings are the following: (1) the most recent model modifications (BLiM-recent), (2) the oldest model modifications (BLiM-oldest), (3) the mean of the modification timespan of the modified model elements (BLiM-mean), and (4) the sum of the modification timespan of the modified model elements (BLiM-sum).

In our evaluation, we have applied our approach to the product models from an industrial partner, BSH. We compare the results of running our BLiM approach with the different fitness functions. We measure the results using the standard information retrieval measurements: recall, precision, and the combination of both (F-measure) [8, 9]. The outcome shows that the use of the most recent modification timespan of a model element as the modification timespan of a model fragment (BLiM-recent) provides the best results, and proves that the approach can be applied in real world environments. The statistical analysis of the results provides evidence of their significance.

The remainder of the paper is structured as follows: in Sect. 2, we present the Domain Specific Language used by the industrial partner. In Sect. 3, we describe our BLiM approach. In Sect. 4, we evaluate our approach with the data provided by the industrial partner. In Sect. 5, we examine the related work of the area. Finally, we present our conclusions in Sect. 6.

2 Background

The running example and the evaluation of this paper are performed through the products of the industrial partner, BSH. In this section, we present the Domain Specific Language (DSL) used by BSH to formalize their products, called IHDSL. In addition, we present the language used by our approach to formalize the model fragments, the Common Variability Language (CVL).

The newest Induction Hobs (IHs) feature full cooking surfaces, where dynamic heating areas are automatically generated and activated or deactivated depending on the shape, size, and position of the cookware placed on the top. In addition, there has been an increase in the type of feedback provided to the user while cooking. All of these changes are being possible at the cost of increasing the complexity of the software behind IHs.

The Domain Specific Language used by BSH to specify the Induction Hobs (IHDSL) is composed of 46 meta-classes, 47 references among them, and more than 180 properties. However, in order to gain legibility and due to intellectual property right concerns, in this paper we use a simplified subset of the IHDSL (see left part of Fig. 1, IHDSL Metamodel and IHDSL Syntax).

Product Model of Fig. 1 depicts an example of a product model specified with the IHDSL. The product model contains four inverters used to power two different inductors. The upper inductor is powered by a single inverter while the lower inductor is powered by the combination of three different inverters. Power managers act as hubs to perform the connection between the inverters and the inductors.

To formalize the model fragments used by the approach we use Common Variability Language (CVL) [10, 11], given its capabilities to formalize a set of model elements as a model fragment. Right part of Fig. 1 shows an example of a model fragment of the product model. The model fragment includes the three

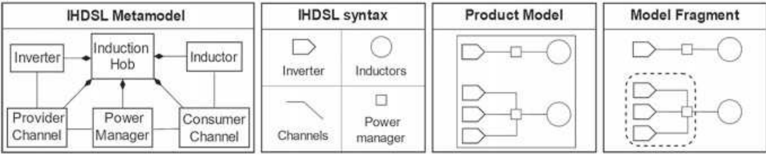


Fig. 1 IHDSL product model and model fragment formalization

inverters in charge of powering the lower inductor along with the three channels and the power manager used to aggregate and manage the power provided by those inverters.

3 Bug Location in Models (BLiM)

This section presents the BLiM approach for bug location. The left part of Fig. 2 shows an example of input for our approach. The approach receives as input a bug description of the bug that the software engineer wants to locate. Typically, these descriptions come from textual documentation of a bug report. Therefore, the query will include some domain specific terms that are similar to those used when specifying the product models. In addition, the software engineer selects a set of product models from the entire family of products that include the bug to be located.

The approach relies on an evolutionary algorithm. The center of Fig. 2 shows a simplified representation of the main steps. The 'Initialize Population' step calculates an initial population of model fragments from the input set of product models. This initial population of model fragments is randomly extracted from the product models. The 'Genetic Operations' produce the new generation of model fragments. First, a selection operation chooses the model fragments that will be used as parents of the new model fragments. The fitness values are used to ensure that the best model fragments are chosen as parents. Then, a crossover operation mixes the model elements of the two parents into a new model fragment. Finally, a mutation operation introduces variations in the new model fragment, in hopes that it achieves better fitness values than its parents. The 'Fitness' step assigns values that assess how good each model fragment is in the following terms: bug description and modification timespan.

As output, the approach provides a list of model fragments that might realize the bug. The output of BLiM (see the right part of Fig. 2) is a ranking of model fragments that realize the target bug. The ranking can be ordered following different criteria, such as the similarity of the model fragments to the bug description, or the model fragment modification timespans.

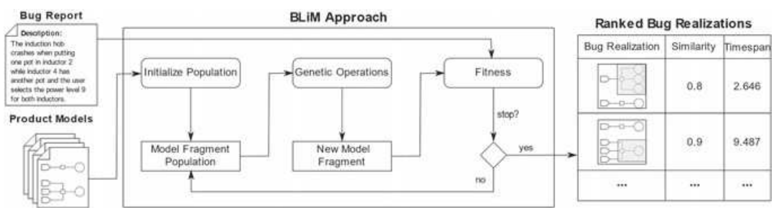


Fig. 2 Overview of the Bug Location Approach in Models: BLiM

3.1 Genetic Operations of the BLiM Approach

The generation of model fragments is performed by applying genetic operators adapted to work on model fragments. In other words, new fragments are generated from existing ones through the use of two genetic operators: the crossover operator, and the mutation operator. We use the crossover and mutation operations presented in [12].

The crossover operation takes a model fragment from a first parent model and the whole product model from a second parent model, generating a new individual that contains elements from both parents and thus preserving the basic mechanics of the crossover operation.

The mutation possibilities of a given model fragment are driven by its associated product model. Each model fragment is associated to a product model, and the model fragment mutates in the context of its associated product model. In other words, the model fragment will gain or drop some elements, but the resulting model fragment will still be part of the referenced product model. For more details about these genetic operations see [12].

3.2 Fitness of the BLiM Approach

In evolutionary algorithms, the fitness step is used to imitate the different degrees of adaptation to the environment that different individuals have. Following this idea, our fitness step is used to determine the suitability of each model fragment to the problem. The input of this step is a population of model fragments, and the produced output is a set with each model fragment from the input population, accompanied by two fitness values: similarity to the feature description, and most recent model fragment modifications.

Model Fragment Similarity to the Bug Description
To assess the relevance of each model fragment in relation to the bug description provided by the user, we apply methods based on Information Retrieval (IR) techniques. Specifically, we apply Latent Semantic Analysis (LSA) [13] to analyze the relationships between the description of the bug provided by the user and the model fragments. There are many IR techniques, but most research efforts show better results when applying LSA [14–16].

LSA constructs vector representations of a query and a corpus of text documents by encoding them as a term-by-document co-occurrence matrix, (i.e., a matrix where each row corresponds to a term and each column corresponds to a document, with the last column corresponding to the query). Each cell holds the number of occurrences of a term (row) inside a document or the query (column). LSA provides good results when applied to source code [14–16]. We use the LSA technique applied to models in the same way as [12].

The documents are text representations of model fragments. The text of the document corresponds to the names and values of the properties and methods of each model fragment (e.g. a model element of the class inductor will contain some properties related to its coil manufacturer and heat potential). The query is constructed from the terms that appear in the bug description. If the textual terms used for the model and the bug description differ too much, the LSA will not work. Therefore, the text from the documents (model fragments) and the text from the query (bug description) are homogenized by applying Natural Language Processing techniques (tokenizing [14], Parts-of-Speech Tagging [17], and Lemmatizing [8]) to eventually reduce this gap. The union of all the words extracted from the documents (model fragments) and from the query (bug description) are the terms (rows) used by our LSA fitness.

We normalize and decompose the matrix into a set of vectors using Singular Value Decomposition (SVD) [13]. One vector that represents the latent semantics of the document is obtained for each model fragment and for the query. Finally, the similarities between the query and each model fragment are calculated as the cosine between the two vectors. The fitness value that is given to each model fragment is the one that we obtain when we calculate the similarity, obtaining values between −1 and 1. For more details see [12].

Timespan Weightings

In this proposed fitness step, the modifications of the model over time are taken into consideration in order to extract the most relevant model for the target bug. In this section, we define the four timespan weighting functions used in our work.

These functions are based on the timespan between the last modification of a model element and the usage day. A recently modified model element (i.e. a short timespan) has a lower timespan value than another model element that was modified farther in the past. As a model fragment is composed by a set of model elements the timespan weighting of the model fragment depends on the timespan weightings of the model elements that compose it.

The timespan is based on the number of days and can therefore be very large when the model fragment was modified a long time ago. To normalize the timespans, mathematical solutions can be used. We used square roots because it has achieved good results in other works that use time differences [11]. The use of square root is more suitable and more effective for the proposed approach.

We devised four objective functions to capture the timespan weightings for the model fragments. Next, we define each of these functions:

The most recent model modifications (recent): this function expresses the concern of capturing primarily the model fragments with the model elements that have the lowest modification timespans. That is, model elements that have been recently modified. Then, the value of the model fragment will be the value of the most recently modified model element. In the example of Fig. 3, the value of the model fragment is 7 days, that means a square root of 2.646.

The oldest model modifications (oldest). This function expresses the concern of capturing primarily the model fragments with the model elements that have the

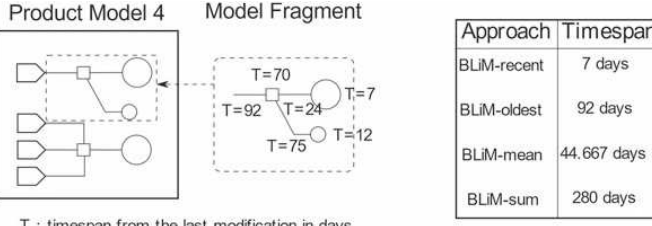


Fig. 3 Timespan of the modifications of the model elements of a fragment

highest modification timespans. That is, model elements that have not been modified for a long time, longer than most other elements. Then, the value of the model fragment will be the value of the model element less recently modified model element. In the example of Fig. 3, the value of the model fragment is 92 days, that means a square root of 9.592.

To avoid taking into account only the extremes (oldest and most recent modifications), we also define these two objective functions:

The mean of the timespan of the modified model elements (mean). This function expresses the concern of capturing primarily the model fragments with the model elements that have the lowest mean timespan. Then, the value of the model fragment will be the value of the mean of the timespan of the modified model elements. In the example of Fig. 3, the value of the model fragment is 46.667 days, that means a square root of 6.831.

The sum of the timespan of the modified model elements (sum). This function expresses the concern of capturing primarily the model fragments with the model elements that have the lowest timespan sum. Then, the value of the model fragment will be the value of the sum of the timespan of the modified model elements. In the example of Fig. 3, the value of the model fragment is 280 days, that means a square root of 16.733.

4 Evaluation: Bug Location in BSH

This section presents the evaluation of our approach: the experimental setup, a description of the case study where we applied the evaluation, the obtained results, the performed statistical analysis, and the threats to validity. To evaluate the approach, we applied it to an industrial case study from our partner, BSH: a leading manufacturer of home appliances in Europe.

4.1 Experimental Setup

The goal of this experiment is the evaluation of the different timespan weighting objective functions as fitness for our BLiM approach. In addition, we compare the BLiM approach with a baseline [18]. The baseline is the approach used in BSH for bug location. It is an evolutionary algorithm guided by textual similarity, however the baseline does not take into account the modification timespan of the model elements. Although it was designed having a more general purpose in mind (Feature Location), it is the best they have for Bug Location in Models.

To evaluate our BLiM approach with the different objective functions (BLiM-recent, BLiM-oldest, BLiM-mean, and BLiM-sum) and the baseline approach, we run each of the approaches and obtain a ranking of model fragments that we can compare with an oracle in order to check accuracy. The inputs of the evaluation process, which are the product family, and bug reports, were provided by BSH.

The oracle is the ground truth, and is used to compare the results provided by the BLiM approach and the baseline. To prepare the oracle, BSH provided us with the bug reports that have occurred in the product models. These bug reports contain natural language bug descriptions and the approved bug realizations. In said bug reports, each bug description is mapped to a single model fragment. A Model fragment is a subset of elements of a model, specified with the model fragment formalization capacities of the CVL [10]. In other words, for each bug, we know which is the associated model fragment that implements it.

The baseline approach is a Single-Objective Evolutionary Algorithm (SOEA), whereas BLiM is Multi-Objective Evolutionary Algorithm (MOEA). The works in [19] shows that common MOEA measures such as hypervolume [20] are not necessarily suitable for comparing solutions by MOEAs (our BLiM approach) with solutions by SOEAs (baseline in this work). Therefore, in order to compare the baseline approach with BLiM, we first take the best solution of the baseline approach for its single-objective (the similarity with the bug description), and then we take the best solution of BLiM with regard to the objective of the baseline approach (the similarity with the bug description), as described in [19]. Finally, these solutions are compared to the bug realization of the oracle in order to get a confusion matrix.

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

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

Then, some performance measurements are derived from the values in the confusion matrix. In particular, we create a report including three performance measurements (recall, precision, and F-measure), for each of the test cases for both BLiM-X and the baseline.

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

$$Recall = \frac{TP}{TP + FN} \quad (1)$$

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

$$Precision = \frac{TP}{TP + FP} \quad (2)$$

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

$$F\text{-}measure = 2 * \frac{Precision * Recall}{Preecision + Recall} = \frac{2 * TP}{2 * TP + FP + FN} \quad (3)$$

Recall values can range between 0% (which means that no single model element from the realization of the bug obtained from the oracle is present in any of the model fragments of the solution) to 100% (which means that all the model elements from the oracle are present in the solution).

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

The approach has been implemented within the Eclipse environment. We used the Eclipse Modeling Framework [21] to manipulate the models and CVL to manage the model fragments. The evolutionary algorithm was built using the Watchmaker Framework for evolutionary Computation [22], which allowed us to implement our own genetic operators. The IR techniques that were used to process

the language were implemented using OpenNLP [23] and the English (Porter2) [24]. The LSI was implemented using the Efficient Java Matrix Library [6].

4.2 Case Study

The case study where we applied our evaluation process is the Induction Hob Product Family of BSH (already presented in Sect. 2 as the running example). The oracle is composed of 46 induction hob models, which are on average composed of more than 500 elements. BSH provided us with documentation of 37 bug reports and the approved bug realizations. For each of the 37 bugs, we created a test case that includes the set of product models where that bug was manifested and a bug description, both obtained from the documentation.

For this case study, we executed 30 independent runs for each of the 37 test cases for BLiM with all the different timespan weightings, and with the baseline (as suggested by [25]), i.e., 37 (bugs) × 5 (approaches) × 30 repetitions = 5550 independent runs.

4.3 Results

In this section, we present the results obtained by both BLiM (with the four fitness functions) and by the baseline approach, for the case study. The 5550 fragments obtained in the executions (one fragment obtained in each execution) had an average size of 52 model elements. Each model element has appeared at least one time in the model fragments.

Table 1 shows the mean values of recall, precision and F-measure of the graphs for both BLiM-X (with the four fitness functions) and the baseline, for the case study. BLiM-recent obtains the best results in recall and precision, providing an average value of 79.10% in recall and 73.26% in precision. The next best results are obtained by BLiM-mean, providing an average value of 71.43% in recall and 64.91% in precision. The third best values are obtained by BLiM-sum, providing an average value of 60.61% in recall and 45.69% in precision. BLiM-oldest obtains the worst value in precision, 27.99%, while the baseline approach obtains the worst

Table 1 Mean values and standard deviations for Recall, Precision and F-measure

	BLiM-recent	BLiM-oldest	BLiM-mean	BLiM-sum	Baseline
Recall ± σ	*79.10 ± 11.75*	51.21 ± 12.58	71.43 ± 11.18	60.61 ± 11.56	44.25 ± 14.79
Precision ± σ	*73.26 ± 9.44*	27.99 ± 7.74	64.91 ± 9.57	45.69 ± 12.45	29.04 ± 9.47
F-measure ± σ	*76.07 ± 8.34*	36.20 ± 7.46	68.02 ± 7.54	52.10 ± 8.26	35.07 ± 9.00

value in recall, 44.25%. In terms of recall and precision, BLiM-recent outperforms the rest of the approaches.

From the results, we can see that there are some bugs (around 24% on average) that are not properly located by the approach. This happens because the fitness function that guides the search is not giving high fitness values to the model fragments realizing those bugs. This can happen due to differences between the language used in the bug descriptions and the product models, or in cases where there are few differences in the modification timespan among the different model fragments.

4.4 Statistical Analysis

To properly compare our BLiM approach (with the four fitness functions) and the baseline approach, all of the data resulting from the empirical analysis was analyzed using statistical methods following the guidelines in [25]. The goals of our statistical analysis are: (1) to provide formal and quantitative evidence (statistical significance) that BLiM-recent does in fact have an impact on the comparison metrics (i.e., that the differences in the results were not obtained by mere chance); and (2) to show that those differences are significant in practice (effect size).

Statistical significance

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

The statistical tests provide a probability value, *p-value*. The *p-value* 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 [25], and so the hypothesis H_0 can be considered false.

The test that we must follow depends on the properties of the data. Since our data does not follow a normal distribution in general, our analysis requires the use of non-parametric techniques. There are several tests for analyzing this kind of data;

Table 2 Holm's post hoc *p-value* and $\hat{A}_{12}$ statistic for each pair of algorithms

	Holm's		$\hat{A}_{12}$	
	Recall	Precision	Recall	Precision
Recent versus oldest	4×10^{-12}	$\ll 2 \times 10^{-16}$	0.9437546	1
Recent versus mean	0.0434	0.019	0.6775018	0.7319211
Recent versus sum	1.1×10^{-6}	7.6×10^{-10}	0.8699781	0.9466764
Recent versus baseline	$\ll 2 \times 10^{-16}$	$\ll 2 \times 10^{-16}$	0.9656684	1
Oldest versus mean	5.8×10^{-7}	$\ll 2 \times 10^{-16}$	0.1278305	0
Oldest versus sum	0.0416	1.4×10^{-6}	0.2885318	0.1022644
Oldest versus baseline	0.0527	0.939	0.6449963	0.4598247
Mean versus sum	0.0078	9.1×10^{-5}	0.7465303	0.8663258
Mean versus baseline	2.7×10^{-11}	$\ll 2 \times 10^{-16}$	0.9181885	1
Sum versus baseline	8.7×10^{-5}	1.3×10^{-6}	0.7991234	0.8363769

however, the Quade test shows that it is the most powerful when working with real data [26]. In addition, according to Conover [27], the Quade test is the one that has shown the best results when the number of algorithms is low (no more than 4 or 5 algorithms).

The *p-value* obtained in the test are $\ll 2 \times 10^{-16}$ for recall and precision, the statistics value obtained are 32.628 and 62.196 for recall and precision respectively. Since the *p-value* are smaller than 0.05 for recall and precision, we reject the null hypothesis. Consequently, we can state that there exist differences among the algorithms (BLiM-recent, BLiM-oldest, BLiM-mean, BLiM-sum, and the baseline) for the performance indicators of recall and precision.

However, with the Quade test, we cannot answer the following question: Which of the algorithms gives the best performance? In this case, the performance of each algorithm should be individually compared against all other alternatives. In order to do this, we perform an additional post hoc analysis. This kind of analysis performs a pair-wise comparison among the results of each algorithm, determining whether statistically significant differences exist among the results of a specific pair of algorithms.

Table 2 shows the *p-value* of Holm's post hoc analysis for the case study and the performance indicators for the five algorithms (BLiM-recent, BLiM-oldest, BLiM-mean, BLiM-sum, and the baseline). The majority of the *p-value* shown in this table are smaller than their corresponding significance threshold value (0.05), indicating that the differences of performance between the algorithms are significant. However, when comparing BLiM-oldest and the baseline (seventh row), the values are greater than the threshold, indicating that the differences between those algorithms could be due to the stochastic nature of the algorithms and are not significant.

Effect size
When comparing algorithms with a large enough number of runs, statistically significant differences can be obtained even if they are so small as to be of no practical value [25]. Then it is important to assess if an algorithm is statistically better than another and to assess the magnitude of the improvement. Effect size measures are needed to analyze this.

For a non-parametric effect size measure, we use Vargha and Delaney's $\hat{A}_{12}$ [20, 26]. $\hat{A}_{12}$ measures the probability that running one algorithm yields higher values than running another algorithm. If the two algorithms are equivalent, then $\hat{A}_{12}$ will be equal to 0.5.

For example, $\hat{A}_{12} = 0.7$ means that we would obtain better results in 70% of the runs with the first algorithm of the pair 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 algorithm of the pair that have been compared. Thus, we have an $\hat{A}_{12}$ value for every pair of algorithms.

Table 2 shows the values of the size effect statistics. In general, the largest differences were obtained between BLiM-recent and the baseline, where BLiM-recent achieves better recall than the baseline 96% of the times and better precision almost all the times. When comparing BLiM-recent and BLiM-mean the differences are not so big, with BLiM-recent outperforming BLiM-mean in recall 67% of the times and in precision 73% of the times.

BLiM-recent obtained the best performance results among the five evaluated approaches (see Table 1). The performed statistical analysis indicated that BLiM-recent outperforms the rest of the approaches in terms of recall and precision (around 70% of the times when compared to BLiM-mean, 90% of the times when compared to BLiM-sum and almost all the times when compared to BLiM-oldest and the baseline). Overall, these results confirm that the use of BLiM-recent against the baseline approach has an actual impact.

4.5 Threats to Validity

In this section, we present some of the threats to the validity of our work. We follow the guidelines suggested by de Oliveira et al. [28] to identify those applicable to this work.

Conclusion validity threats: To address the *not accounting for random variation* threat, we considered 30 independent runs for each bug with each algorithm. As we used the approach that BSH uses for bug location as a comparison baseline, the *lack of a meaningful comparison baseline* threat is addressed. In this paper we employed standard statistical analysis following accepted guidelines [25] to avoid the *lack of a formal hypothesis and statistical tests* threat. For avoid the *lack of a good descriptive analysis* threat, we have used the precision, recall, and F-measure measurements to analyze the confusion matrix obtained from the experiments; however, other measurements could be applied.

Internal validity threats: To address the *poor parameter settings* threat, we used standard values for the algorithms. As suggested by Arcuri and Fraser [25], default values are good enough to measure the performance of location techniques in the context of testing. Nevertheless, we plan to evaluate all the parameters of our algorithm in a future work. As we have evaluated our work in an industrial case study the *lack of real problem instances* threat is addressed.

Construct validity threats: To address the *lack of assessing the validity of cost measures* threat, we performed a fair comparison among BLiM-X and the baseline by generating the same number of model fragments and using the same number of fitness evaluations.

External validity threats: The *lack of a clear object selection strategy* and the *lack of evaluations for instances of growing size and complexity* threats are addressed by using an industrial case study, BSH. Our instances are collected from real world problems. In addition, regarding the generalization of our approach, the set of models where the bugs have to be located are conforming to MOF (the OMG metalanguage for defining modeling languages), and the bug reports must be provided using natural language. Then, our evaluation does not rely on the particular conditions of our domain. Nevertheless, the evaluation should be replicated in other domains before assuring their generalization.

5 Related Work

Saha et al. [15] presented BLUiR, which uses a baseline "TF.IDF model". They believe that code constructs improve the accuracy of bug localization. They syntactically parse the source code into four document fields: class, method, variable, and comment. The summary and the description of a bug report are considered as two query fields. Textual similarities are computed for each of the eight-document field-query field pairs and then summed up into an overall ranking measure. Kim et al. [29] propose both a one-phase and a two-phase prediction model to recommend files to fix. In the one-phase model, they create features from textual information and metadata of bug reports, apply Naïve Bayes to train the model using previously fixed files as classification labels, and then use the trained model to assign multiple source files to a bug report. In the two-phase model, they first apply their one-phase model to classify a new bug report as either "predictable" or "deficient", and then make predictions only for "predictable" reports. Unlike us, all of these approaches do not take into account the modification timespan of the retrieved source locations. Furthermore, these approaches target code while our approach targets models to locate the bug realizations.

Zamani et al. [30] proposed an approach that included weighting and ranking the source code locations based on both the textual similarity with a change request and the use of the time metadata. This approach gives better results than IR techniques. However, their approach is applied at the source code level, while we use a Multi-Objective Evolutionary Algorithm to address the location of bugs in models.

In addition, other approaches use genetic algorithms to locate features in models, Font et al. [12, 18] propose two approaches to locate features in a model. However, these works do not take into account the modification timespan of the model elements. Our work, in contrast, is focused on searching bug realizations, hence, the timespan weighting is an important piece of the approach in order to obtain accurate results.

6 Conclusion

Bug Location is a significant maintenance activity. In this paper, we have proposed four approaches for bug location in models (BLiM-recent, BLiM-oldest, BLiM-mean, BLiM-sum) and compared them with a baseline. Our BLiM-X approaches, in order to guide our bug location evolutionary algorithm, consider: (1) the similitude to the bug description, and (2) the modification timespan weightings of the models.

We evaluate which approach produces better results in terms of precision, recall and F-measure. To do so, we applied the five approaches in an industrial domain, BSH, that has a model based product family (firmware of Induction Hobs). We report our evaluation, including: experimental setup, results, statistical analysis, and threats to validity.

The results show that the application of the Defect Localization Principle that has achieved good results in bug location in code leads to a significant improvement when it has applied to bug location in models compared to the baseline approach. The findings of our work are:

- The application of the Defect Localization Principle using the most recent modification timespan of a model element (BLiM-recent) provides the best results in our study.
- Results also show that our approach can be applied in real world environments. Nevertheless, we need further experiments that involve the final users of our approach in order assure that can be applied in all real world environments.

In addition, this work presents a statistical analysis of the results. This analysis provides evidence of the significance of results obtained, and we can state that they were not obtained by mere chance.

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

References

1. Lehman, M.M., Ramil, J.F., Kahen, G.: A paradigm for the behavioural modelling of software processes using system dynamics. Citeseer (2001)
2. Antoniol, G., Gueheneuc, Y.-G.: Feature identification: an epidemiological metaphor. IEEE Trans. Softw. Eng. **32**(9), 627–641 (2006)
3. Dit, B., Revelle, M., Gethers, M., Poshyvanyk, D.: Feature location in source code: a taxonomy and survey. J. Softw. Maint. Evol. Res. Pract. (2011)
4. Rubin, J., Chechik, M.: A survey of feature location techniques. In: Reinhartz-Berger, I., Sturm, A., Clark, T., Cohen, S., Bettin, J. (eds.) Domain Engineering, pp. 29–58. Springer, Berlin (2013)
5. Hassan, A.E., Holt, R.C.: The top ten list: dynamic fault prediction. In: 21st IEEE International Conference on Software Maintenance (ICSM'05), pp. 263–272 (2005)
6. Efficient Java Matrix Library, https://ejml.org. Accessed: April 07, 2016 (2016)
7. Sisman, B., Kak, A.C.: Incorporating version histories in Information Retrieval based bug localization. In: 2012 9th IEEE Working Conference on Mining Software Repositories (MSR), pp. 50–59 (2012)
8. Plisson, J., Lavrac, N., Mladenic, D.: A rule based approach to word lemmatization. In: Proceedings of the 7th International Multi-Conference Information Society IS 2004. pp. 83–86 (2004)
9. Vargha, A., Delaney, H.D.: A critique and improvement of the CL common language effect size statistics of McGraw and Wong. J. Educ. Behav. Stat. **25**(2), 101–132 (2000)
10. Haugen, Ø., Møller-Pedersen, B., Oldevik, J., Olsen, G.K., Svendsen, A.: Adding standardized variability to domain specific languages. In: Proceedings of the 2008 12th International Software Product Line Conference, pp. 139–148. IEEE Computer Society, Washington, DC, USA (2008)
11. Zimmermann, T., Weisgerber, P., Diehl, S., Zeller, A.: Mining version histories to guide software changes. In: Proceedings of the 26th International Conference on Software Engineering, pp. 563–572. IEEE Computer Society, Washington, DC, USA (2004)
12. Font, J., Arcega, L., Haugen, Ø., Cetina, C.: Feature location in model-based software product lines through a genetic algorithm. In: 15th International Conference on Software Reuse, Limassol, Cyprus (2016)
13. Landauer, T.K., Foltz, P.W., Laham, D.: An introduction to latent semantic analysis. Discourse Process **25**(2–3), 259–284 (1998)
14. Manning, C.D., Schütze, H.: Foundations of Natural Language Processing. Reading, 678 (2000)
15. Saha, R.K., Lease, M., Khurshid, S., Perry, D.E.: Improving bug localization using structured information retrieval. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 345–355 (2013)
16. Salton, G., McGill, M.J.: Introduction to Modern Information Retrieval. McGraw-Hill Inc., New York, NY, USA (1986)
17. Hulth, A.: Improved automatic keyword extraction given more linguistic knowledge. In: Proceedings of the 2003 Conference on Empirical Methods in Natural Language Processing, pp. 216–223. Association for Computational Linguistics, Stroudsburg, PA, USA (2003)
18. Font, J., Arcega, L., Haugen, Ø., Cetina, C.: Feature location in models through a genetic algorithm driven by information retrieval techniques. In: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, pp. 272–282. ACM, New York (2016)
19. Ishibuchi, H., Nojima, Y., Doi, T.: Comparison between single-objective and multi-objective genetic algorithms: performance comparison and performance measures. In: 2006 IEEE International Conference on Evolutionary Computation, pp. 1143–1150 (2006)
20. Zitzler, E., Thiele, L.: Multiobjective optimization using evolutionary algorithms—a comparative case study. In: Eiben, A.E., Bäck, T., Schoenauer, M., Schwefel, H.-P. (eds.)

Parallel Problem Solving from Nature—PPSN V: 5th International Conference Amsterdam, The Netherlands September 27–30, 1998 Proceedings. pp. 292–301. Springer Berlin Heidelberg, Berlin, Heidelberg (1998)
21. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: Eclipse Modeling Framework, 2nd edn. Addison-Wesley Professional (2009)
22. Dyer, D.W.: The watchmaker framework for evolutionary computation (evolutionary/genetic algorithms for Java) (2006)
23. Apache OpenNLP: Toolkit for the processing of natural language text. http://opennlp.apache.org/. Accessed: April 07, 2016 (2010)
24. The English (Porter2) stemming algorithm. http://snowball.tartarus.org/algorithms/english/stemmer.html, Accessed: April 07, 2016 (2002)
25. Arcuri, A., Fraser, G.: Parameter tuning or default values? An empirical investigation in search-based software engineering. Empir. Softw. Eng. **18**(3), 594–623 (2013)
26. García, S., Fernández, A., Luengo, J., Herrera, F.: Advanced nonparametric tests for multiple comparisons in the design of experiments in computational intelligence and data mining: experimental analysis of power. Inf. Sci. (NY) **180**(10), 2044–2064 (2010)
27. Conover, W.J.: Practical Nonparametric Statistics, 3rd edn. Wiley, New York (1999)
28. de Oliveira Barros, M., Dias-Neto, A.C.: 0006/2011-Threats to validity in search-based software engineering empirical studies. RelaTe-DIA 5(1) (2011)
29. Kim, D., Tao, Y., Kim, S., Zeller, A.: Where should we fix this bug? A two-phase recommendation model. IEEE Trans. Softw. Eng. **39**(11), 1597–1610 (2013)
30. Zamani, S., Lee, S.P., Shokripour, R., Anvik, J.: A noun-based approach to feature location using time-aware term-weighting. Inf. Softw. Technol. **56**(8), 991–1011 (2014)

13.2 MODELS'18 Paper

Title: Evolutionary Algorithm for Bug Localization in the Reconfigurations of Models at Runtime.

Authors: Lorena Arcega, Jaime Font, Carlos Cetina.

Proceedings: Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS '18).

Location: Copenhagen, Denmark - October 14-19, 2018

Publisher: ACM

Pages: 90-100

DOI: `https://doi.org/10.1145/3239372.3239392`

Contribution: Lorena Arcega is the main author of the paper and is responsible for 90% of the work. She was also responsible for the oral presentation of the work which took place during the conference.

Evolutionary Algorithm for Bug Localization in the Reconfigurations of Models at Runtime

Lorena Arcega
SVIT Research Group
Universidad San Jorge
Zaragoza, Spain
Department of Informatics
University of Oslo
Oslo, Norway
larcega@usj.es

Jaime Font
SVIT Research Group
Universidad San Jorge
Zaragoza, Spain
Department of Informatics
University of Oslo
Oslo, Norway
jfont@usj.es

Carlos Cetina
SVIT Research Group
Universidad San Jorge
Zaragoza, Spain
ccetina@usj.es

ABSTRACT

Systems with models at runtime are becoming increasingly complex, and this is also accompanied by more software bugs. In this paper, we focus on bugs appearing as the result of dynamic reconfigurations of the system due to context changes. We materialize our approach for bug localization in reconfigurations as an evolutionary algorithm. We guide the evolutionary algorithm with a fitness function that measures the similarity to the description of the bug report. The result is a ranked list of reconfiguration sequences, which is intended to identify the reconfiguration rules that are relevant to the bug. We evaluated our approach in BSH and CAF, two real-world industrial case studies, measuring the results in terms of recall, precision, F-measure and Matthews Correlation Coefficient (MCC). In our evaluation, we compare our approach with two other approaches: a baseline that is the one used by our industrial partners for bug localization and a random search as sanity check. Our study shows that our approach, which takes advantage of the reconfigurations of models at runtime, outperforms the other two approaches. We also performed a statistical analysis to provide evidence of the significance of the results.

CCS CONCEPTS

• **Software and its engineering → Model-driven software engineering**; **Search-based software engineering**;

KEYWORDS

Bug Localization, Models at Runtime

ACM Reference Format:
Lorena Arcega, Jaime Font, and Carlos Cetina. 2018. Evolutionary Algorithm for Bug Localization in the Reconfigurations of Models at Runtime. In *ACM/IEEE 21th International Conference on Model Driven Engineering Languages and Systems (MODELS '18), October 14–19, 2018, Copenhagen, Denmark*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3239372.3239392

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MODELS '18, October 14–19, 2018, Copenhagen, Denmark
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-4949-9/18/10…$15.00
https://doi.org/10.1145/3239372.3239392

1 INTRODUCTION

Model-Driven Engineering (MDE) is being applied in an ever increasing manner to cope with the complexity of software systems by raising the level of abstraction [23]. Models at runtime [6] is defined as *a causally connected self-representation of the associated system that emphasizes the structure, behavior, or goals of the system from a problem space perspective* [7]. A recent survey [36] has classified the objective of the use of models at runtime in the following categories: adaptation [27, 35], monitoring, simulation and prediction [12, 16, 24], abstraction and platform independence [38], consistency and conformance [2], policy checking and enforcement [34], and error handling [10].

Software is becoming increasingly complex, and systems with models at runtime are not an exception. Unfortunately, an increase in complexity is accompanied by an increase in the appearance of software bugs. Hence, software maintenance is becoming more and more important. Lehman et al. [19] 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 to make changes correctly.

In a system with models at runtime, the models experience reconfigurations at runtime due to context changes, being these reconfigurations a source of bugs. A recent Search-Based Software Engineering survey [40] reveals that none of the Bug Location approaches take in account the bugs caused by the reconfigurations of a models at runtime system.

Our work is focused on locating bugs that appear as the result of dynamic reconfigurations of the system due to context changes. In this paper, we present an approach for bug localization in the reconfigurations that occur in runtime models called EBRo. We materialize our approach for bug localization in reconfigurations through an evolutionary algorithm. The solutions provided by our approach are sequences of reconfigurations that, when followed, might lead to the model at runtime which contains the located bug.

The evolutionary algorithm is guided by a fitness function that considers the similarity to the description of the bug report. To measure the textual similarity, we start from an initial model at runtime to which we apply a sequence of reconfigurations, obtaining another model in which we evaluate whether the modified elements are similar with the description of the bug. As a result, software engineers obtain a ranked list of sequences of reconfigurations,

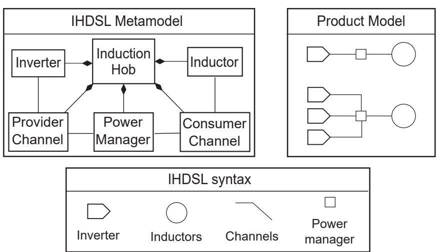


Figure 1: IHDSL metamodel, syntax, and product model

intended to identify the reconfiguration rules that are relevant to the bug.

We have applied our approach to the industrial domains of BSH and CAF. BSH is one of the larges manufacturers of home appliances in Europe. Its induction division has been producing induction hobs under the brands of Bosch and Siemens for the last 15 years. CAF produces a family of Programmable Logic Controller (PLC) software to manage the trains they manufacture, which has been under development for more than 25 years. The firmware from both industries is specified by means of Domain Specific Languages. The firmware of the products is generated from the DSL models and uses models at runtime to change the configuration when their products are in operation.

In our evaluation, we compare our approach with a baseline approach. Since there are no specific baselines for bug localization in reconfigurations of models at runtime, we use as a baseline the approach used by BSH and CAF for bug localization. In addition, we compare our approach with a random search approach (RS) as sanity check. We apply our EBRo approach, the baseline approach and the RS approach to the product families of BSH and CAF. They provided us with documentation about bugs. For each bug, the documentation provided a bug description, the reconfigurations that trigger the bug and the localization of the bug. Taking the bug descriptions, the set of reconfigurations, and an initial model of the product family as input, and the reconfigurations that trigger the bug and the location of the bugs as oracle (ground truth), we measure the results in terms of the standard measurements accepted by the software engineering community: recall, precision, F-measure (the combination of both recall and precision), and Matthews Correlation Coefficient (MCC) [25, 33].

EBRo performed better than the other algorithms in terms of the four measured performance indicators. On average, up to 78.72% of the reconfigurations that were expected to trigger the bugs being located (according to the oracle) were found when EBRo was used (up to 66.84% for the baseline, and 38.42% for the random search approach). It turns out that the genetic operations performed by the EBRo approach with the fitness function are able to properly traverse search spaces originated when locating bugs over runtime reconfigurations of the system.

The remainder of the paper is structured as follows. In Section 2, we present the Domain Specific Language used by one of our industrial partner. In Section 3, we explain the motivation for bug localization in reconfiguration rules. In Section 4, we describe our bug localization approach. In Section 5, we evaluate the application of our approach in BSH and CAF. In Section 6, we examine the related work of the area. Finally, we present our conclusions in Section 7.

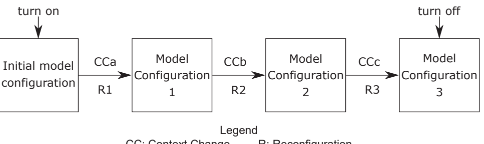


Figure 2: Induction Hob at runtime

2 BACKGROUND

In this section, we present the Domain Specific Language (DSL) that is used by BSH to formalize their products and the models at runtime of the induction hobs to which we apply our approach.

The newest Induction Hobs (IHs) [1] feature full cooking surfaces, where dynamic heating areas are automatically generated and activated or deactivated depending on the shape, size, and position of the cookware placed on the top. These dynamic areas are managed at runtime by calculating the resulting model after the changes in the context of the IH.

The Domain Specific Language used by our industrial partner to specify the Induction Hobs (IHDSL) is composed of 46 metaclasses, 47 references among them, and more than 180 properties. For legibility reasons and due to intellectual property right concerns, in this section, we show a simplified subset of the IHDSL (see Fig. 1, IHDSL Metamodel and IHDSL Syntax). However, the evaluation was performed using the full IHDSL that is used in BSH. The Product Model in Figure 1 depicts an example of a product model that is specified with the IHDSL.

Inverters are in charge of transforming the electric supply to match the specific requirements of the IH. Then, the energy is transferred to the inductors through the channels. There can be several alternative channels, which enable different heating strategies depending on the cookware placed on top of the IH at runtime. 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.

Figure 2 shows the behavior of an Induction Hob at runtime [2]. The IH is turned on in an initial configuration with a known model. In the face of changes in the context (CCs in Figure 2), reconfigurations (Rs in Figure 2) are triggered in order to change the configuration of the IH. Then, the Induction Hob is in a different configuration and therefore in a different model (Model Configurations in Figure 2). Some examples of relevant context changes include putting a pot on top, the pot reaches the set temperature, the pot is moved to other place on the IH, or liquid spills from the pot onto the surface. The reconfigurations activate or deactivate inductors and inverters and connect them through channels.

[1] https://www.youtube.com/watch?v=HjZ_nB-TY7w
[2] https://www.youtube.com/watch?v=Gp6urUZZbek

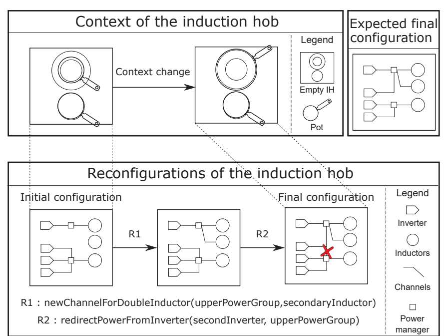


Figure 3: Example of a context change, a bugged reconfiguration and the reconfiguration rules performed

3 MOTIVATION

Figure 3 shows a reconfiguration that occurs in the Induction Hobs of BSH. In the initial configuration, the Induction Hob has two pots on top, heated through two inductors. The upper inductor is powered with one inverter and the bottom inductor is powered with three inverters. When a bigger pot is placed in the upper inductor, the Induction Hob reconfigures itself. Another inductor is activated ('R1'), and more energy is needed to heat the pot ('R2'). Therefore, the second inverter should give power to the upper inductors. However, the second inverter is not disconnected from the bottom power group (see the cross in Figure 3). This situation causes a bug, because when the user changes the power level of the upper inductors, the same power level will be applied to the bottom inductors, and vice versa. This bug was solved by modifying the reconfiguration rule 'R2' so that in addition to redirecting the power of the inverter, the inverter is also disconnected from the previous power group (see expected final configuration in Figure 3).

While the system is in use, any reconfiguration can be activated at any time. An Induction Hob can have a useful life of more than ten years, so the number of different combinations of reconfigurations that can be triggered is very high. In addition, some commercial Induction Hobs can have up to 48 inductors that dynamically connect and disconnect from the inverters. This Induction Hob has 2^{48} possibilities for activating or deactivating the inductors, beside differences in activation and deactivation orders. In addition, inductors and inverters are not the only dynamic components of the Induction Hob. Since the search space is too large, it is impossible to explore the space of possibilities exhaustively. Therefore, we use an evolutionary algorithm to locate bugs in the reconfiguration rules.

4 BUG LOCALIZATION IN RECONFIGURATION RULES

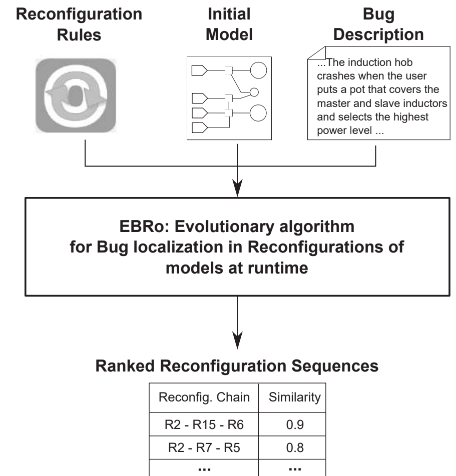
This section describes how the issue of bug localization in the reconfigurations of runtime models can be addressed by using our evolutionary algorithm, and the principles of our proposed method. Therefore, we first present an overview of our approach and, subsequently, provide the details of the approach and our adaptation of the evolutionary algorithm.

4.1 Approach Overview

The general structure of our approach (EBRo) is introduced in Figure 4. The goal of EBRo is to obtain a ranked list of sequences of reconfigurations rules from a given list of reconfiguration rules that may trigger the bug specified by the bug description. Our EBRo approach takes the following inputs:

- A set of reconfiguration rules that describe the changes in the model at runtime. The reconfiguration rules are triggered by context changes.
- An initial model which is the model that specifies the initial configuration. In our case, the configuration when the induction hob is turned on.
- A bug description of the target bug, using natural language. Typically, the description comes from textual documentation of a bug report. Therefore, the query will include some domain specific terms that are similar to those used when specifying the reconfiguration rules and the models. The knowledge of the engineers about the domain and the reconfiguration rules and the models will be useful for selecting the description from the bug report.

The output of EBRo (see Figure 4) is an ordered set of reconfiguration sequences that might trigger the target bug. The ranking is ordered following the similarity to the bug description. The search space for our approach is determined not only by the number of triggered reconfigurations, but also by the order in which they are applied. To explore the search space, EBRo uses an evolutionary



Reconfig. Chain	Similarity
R2 - R15 - R6	0.9
R2 - R7 - R5	0.8
...	...

Figure 4: Input and output of our bug localization in reconfigurations of models at runtime

algorithm that enables the exploration of a large number of possible reconfigurations. The evolutionary algorithm and its adaptation to the bug localization problem are described in the following sections.

4.2 Adapting the Evolutionary Algorithm for Bug Localization in Reconfigurations of Models at Runtime

Evolutionary algorithms are inspired by Darwin's evolutionary theory, where a population of individuals is modified through crossover and mutation operators [9]. Hence, to develop an evolutionary algorithm, the following elements have to be defined:

- Representation of the individuals.
- Evaluation of the individuals using a fitness function for each objective to determine a quantitative measure of their ability to solve the problem under consideration.
- Selection of the individuals to transmit from one generation to another.
- Creation of new individuals using genetic operators (crossover and mutation) to explore the search space.

The following paragraphs describe the design of these elements for our evolutionary algorithm for bug localization in reconfigurations of models at runtime.

4.2.1 Individual representation. To represent a candidate solution (individual), we used a vector representation. Each vector's dimension represents a reconfiguration rule. Thus, a solution is defined as a sequence of reconfigurations applied to a model. The size of the solution represents the number of reconfigurations (dimensions) in the vector. When created, the order of the reconfigurations corresponds to their positions in the vector.

R1: newChannelForDoubleInductor(upperPowerGroup, secondaryInductor)
R21: redirectPowerFromInverter(secondInverter,upperPowerGroup)
R3: activatePowerFromInverter(secondInverter)

Figure 5: Representation of an individual

An example of an individual is given in Figure 5. This individual contains three dimensions that correspond to three reconfigurations applied to the initial model. For instance, the predicate *newChannelForDoubleInductor(upperPowerGroup, secondaryInductor)* means that a new *channel* is created in the *upper power group*, connecting it with the *secondary inductor*.

4.2.2 Fitness function. After creating a solution, it should be assessed using a fitness function. The fitness function quantifies the quality of the proposed reconfiguration sequence. In our work, we use an information retrieval technique called Latent Semantic Indexing (LSI). Our algorithm assesses the relevance of each reconfiguration sequence in relation to the bug description provided by the user. The input of this step is a set of reconfiguration sequences, and the output is the set of reconfiguration sequences, where each reconfiguration sequence has been assigned with a fitness value regarding its similarity to the bug description.

To assess the relevance of each reconfiguration sequence in relation to the bug description provided by the user, we apply methods based on Information Retrieval (IR) techniques. Specifically, we apply Latent Semantic Indexing (LSI) [20, 32] to analyze the relationships between the description of the bug provided by the user and the reconfiguration sequences. There are many IR techniques, but most of the efforts show better results when applying LSI [20, 30, 32], specially when working with source code. Models are representations at a higher abstraction level than the source code, and the language used to build them is closer to the bug description language; therefore, we expect it to work better than when applied to source code.

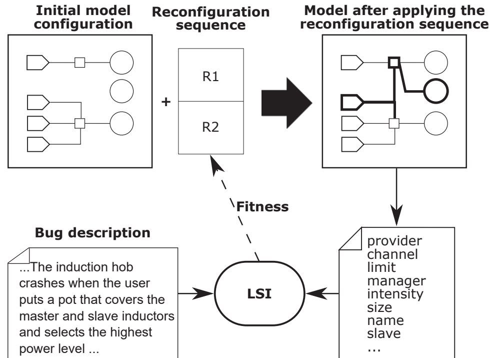


Figure 6: Terms extraction from a reconfiguration sequence

LSI constructs vector representations of a query and a corpus of text documents by encoding them as a term by document co-occurrence matrix, (i.e., a matrix where each row corresponds to terms and each column corresponds to documents, with the last column corresponding to the query). Each cell holds the number of occurrences of a term (row) inside a document or the query (column).

Figure 6 shows how we extract the texts needed to use LSI. First, we apply the reconfiguration sequence to the initial model configuration. After applying it, we obtain a new model from which we extract the model elements that have been modified by the reconfigurations. In Figure 6, the modified model elements are the ones in bold. The texts for the LSI documents are the names and values of the properties and methods of each model element.

In our work, the LSI documents are model elements, i.e., a document of text is generated from the text of the model elements that have been modified by the reconfiguration. The query is constructed from the text that appears in the bug description. If the terms used for the model elements and the bug description differ too much, the LSI will not work. Therefore, the text from the documents (model elements) and the text from the query (bug description) are homogenized by applying well-known Natural Language Processing techniques (tokenizing, Parts-of-Speech Tagging, and Lemmatizing) to reduce this gap. If the languages used differ too much, other

techniques such as manual annotation of the model elements could be applied at the expense of increasing the effort.

The union of all the keywords extracted from the documents (model elements) and from the query (bug description) are the terms (rows) used by our LSI fitness. Each column is one of the model elements that have been modified by the reconfiguration. The last column is the query obtained from the bug description of the user. Each row is one of the terms extracted from the corpuses of text composed by all of the model elements and the query itself. Each cell has the number of occurrences of each of the terms in the model elements.

Once the matrix is built, we normalize and decompose it into a set of vectors using a matrix factorization technique called Singular Value Decomposition (SVD) [18]. One vector that represents the latent semantics of the document is obtained for each model fragment and the query. Finally, the similarities between the query and each model fragment are calculated as the cosine between the two vectors. The fitness value that is given to each model fragment is the one that we obtain when we calculate the similarity, obtaining values between -1 and 1.

4.2.3 Selection. To select individuals, we use stochastic universal sampling (SUS) [5]. This technique of selection of an individual is directly proportional to its relative fitness in the population. SUS is a random selection algorithm which gives a higher probability of selection to the fittest solutions while still giving a chance to every solution.

In each iteration of the algorithm, SUS is used to select individuals from the population (P_n) for the next generation of the population (P_{n+1}). The selected individuals will be the ones that generate the next individuals using genetic operations.

4.2.4 Genetic operators. To better explore the search space, the crossover and mutation operators are defined:

- Crossover: we use a single, random, cut-point crossover. It starts by selecting and splitting at random two parent solutions. When two parent individuals are selected, a random cut point is determined to split them into two sub-vectors. Then, the crossover creates two child solutions by putting, for the first child, the first part of the first parent with the second part of the second parent, and, for the second child, the first part of the second parent with the second part of the first parent.
  Each solution has a length limit in terms of number of reconfigurations. When applying the crossover operator, the new solution may have the minimum length between the two parents. Then, the crossover operator must enforce the length limit constraint by eliminating some reconfiguration rules.
  Figure 7 shows an example of applying the crossover operator. In this example, Parent 1 (P_1) and Parent 2 (P_2) are combined to generate two new solutions. The upper sub-vector of P_1 is combined with the bottom sub-vector of P_2 to form Child 1, and the bottom sub-vector of P_1 is combined with the upper sub-vector of P_2 to form Child 2.
- Mutation: This operator consists of randomly changing one or more reconfigurations in the vector of reconfigurations.
  Given an individual, the mutation operator first randomly selects some positions in the vector representation of the individual. Then, the selected dimensions are replaced by other reconfiguration rules.
  Figure 7 shows an example of applying the mutation operator. In Child 1, the mutation operator replaces dimension number two (*R5* by *R12*), while in Child 2, the mutation operator replaces dimensions number one and three (*R14* and *R3* by *R2* and *R6*).

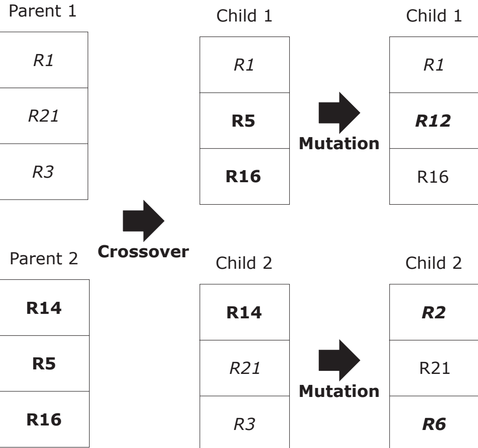


Figure 7: Crossover and mutation operators applied to reconfigurations

When creating the sequence of reconfigurations, we do not guarantee that they are feasible and that they can be applied. However, this could be solved by applying some repair operations that are out of the scope of this paper.

As a result, new reconfiguration sequences are created. In other words, the new reconfiguration sequences represent other possible solutions that can trigger the bug for the specific bug being located.

Overall, the aim of the approach is to find the most relevant reconfiguration sequence that triggers the bug described by the bug report. To do so, the algorithm of EBRo performs a search guided by a fitness function. This search is done among the different reconfiguration sequences (previously obtained by applying the mutation and crossover operations) that could conform to the bug description.

5 EVALUATION

This section presents the evaluation of our approach: the oracle preparation, the experimental setup, the description of the case studies where we applied the evaluation, the obtained results, the statistical analysis, the discussion of the results, and the threats to the validity of our work.

To evaluate the approach, we applied it to two long-living industrial case studies from two of our partners: BSH, the leading manufacturer of home appliances in Europe; and CAF, an international provider of railway solutions all over the world.

5.1 Oracle preparation

The oracle is the ground truth and is used to compare the results provided by the EBRo approach, the baseline, and a random search (RS) that works as sanity check. The baseline is the approach used by our industrial partner for bug localization [14]. A bug can be seen as an unwanted functionality, and a feature represents a functionality. For this reason, the feature localization approach presented in Font et. al [14] can be used for bug localization. Even though it was designed with a more general purpose in mind (feature localization), said approach is the best bug localization technique available to our industrial partner.

To prepare the oracle, our industrial partner provided us with the bug reports associated to bugs that have occurred in the product models. These bug reports contain natural language bug descriptions, the approved reconfigurations that trigger the target bugs and the model fragments that contain the bugs.

5.2 Experimental setup

This experiment evaluates whether or not the information found in the reconfiguration sequences improves the bug localization results. In addition, we compare the EBRo approach with the baseline [14] and with a random search (RS) sanity check. If RS outperforms an intelligent search method, we can conclude that there is no need to use a metaheuristic search.

The inputs of the evaluation process provided by our industrial partner are the models of the induction hobs, the entire set of reconfiguration rules, and the bug reports. The models, reconfiguration rules, and bug descriptions are used to run the EBRo and RS approaches, while the models and descriptions are used to run the baseline approach. We run each of the approaches and obtain a ranking of solutions that we can compare with an oracle in order to check accuracy. From the EBRo and RS approaches, we obtain reconfiguration sequences as solutions while from the baseline, we obtain model fragments as solutions.

Therefore, in order to compare the baseline and RS approaches against EBRo, we take the best solutions from the three approaches and compare them to the actual solution (from the oracle) that contains the trigger of the target bug in order to get a confusion matrix.

A confusion matrix is a table that is often used to describe the performance of a classification model (in this case, EBRo, the baseline, and RS) on a set of test data (the solutions) for which the true values are known (from the oracles). In our case, each solution that is output by the EBRo and the RS approaches is a sequence of reconfigurations composed of a subset of reconfigurations that are present in the sequence that triggers the bug. Since the granularity will be at the level of reconfigurations, the presence or absence of each reconfiguration rule will be considered as a classification. In the same way, each solution outputted by the baseline approach is a model fragment composed of a subset of the model elements that are present in the product model (where the bug is being located). Since the granularity will be at the level of model elements, the presence or absence of each model element will be considered as a classification. Therefore, our confusion matrices will distinguish between two values: TRUE (presence), and FALSE (absence).

The comparison process contrasts a result from one of the evaluated approaches with the ground truth from the oracle. We obtain a confusion matrix for each of the solutions predicted by each of the approaches. The confusion matrix arranges the results of the comparison into four categories:

- True positive (TP): an element present in the predicted solutions that is also present in the actual solution,
- True Negative (TN): an element not present in the predicted solution that is not present in the actual solution,
- False Positive (FP): an element present in the predicted solution that is not present in the actual solution, and
- False Negative (FN): an element not present in the predicted solution that is present in the actual solution.

The confusion matrix holds the results of the comparison between the predicted results and the actual results. The result of the sum of all the categories (TP+TN+FP+FN) is the number of dimensions (n) of the vector that contains the predicted solution. However, in order to evaluate the performance of the approach, it is necessary to extract some measurements from the confusion matrix. Therefore, some performance measurements are derived from the values in the confusion matrix. Specifically, we create a report that includes four performance measurements (recall, precision, F-measure, and MCC) for each of the test cases for EBRo, the baseline, and the RS approach.

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

$$Recall = \frac{TP}{TP + FN} \tag{1}$$

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

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

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

$$F{-}measure = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{3}$$

Recall values can range between 0% (i.e., no single element from the actual solution is present in the predicted solution) to 100% (i.e., all the elements from the actual solution are present in the predicted solution).

Precision values can range between 0% (i.e., no single element from the predicted solution is present in the actual solution) to 100% (i.e., all the element from the predicted solution are present in the actual solution). A value of 100% precision and 100% recall implies that both the predicted solution and the actual solution are the same.

However, none of these measures correctly handle negative examples (TN). MCC is a correlation coefficient between the observed

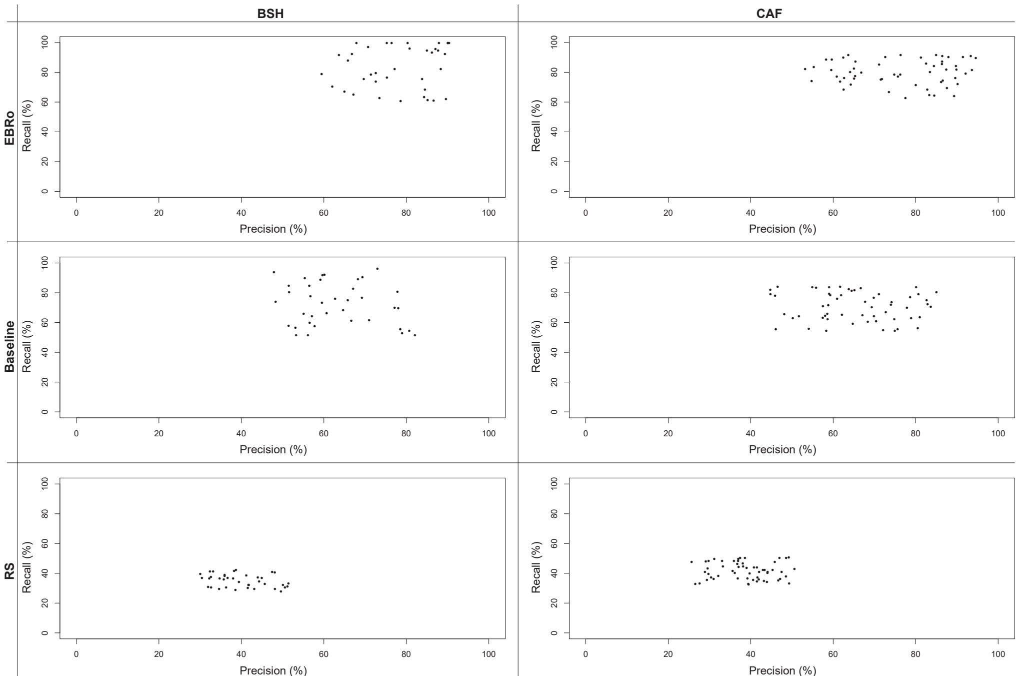


Figure 8: Mean recall and precision for EBRo, the baseline and the RS approaches in BSH and CAF

and predicted binary classifications that takes into account all of the observed values (TP, TN, FP, FN). MCC is a balanced measure which can be used even if the search space and the predicted solution are of very different sizes [8]. For this reason, MCC is one of the best measures for describing a confusion matrix [31]. It is defined as follows:

$$MCC = \frac{TP * TN - FP * FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \quad (4)$$

5.3 Case studies

5.3.1 BSH. The first case study where we apply our evaluation process is BSH (already presented in Section 2). The oracle is composed of 46 induction hob models and its revisions over time where, on average, each product model is composed of more than 500 elements. Our industrial partner provided us with documentation of 37 bug reports, the approved reconfiguration sequences that triggers the bugs and the model fragments that contain the bugs. For each of the 37 bugs, we created a test case that includes the initial model, the set of reconfiguration rules and the model fragment where that bug was manifested and a bug description, all obtained from the documentation.

For this case study, we executed 30 independent runs for each of the 37 test cases for each approach (as suggested by [4]), i.e., 37 (bugs) * 3 (approaches) * 30 repetitions = 3330 independent runs.

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

The DSL of our industrial partner has the required expressiveness to describe the interaction between the main pieces of equipment installed in a train unit and their reconfigurations at runtime. Examples of reconfigurations at runtime can be coupling (when two trains are coupled together to increase the transport capacity or to

rescue a train that is damaged) or converter assistance (allow the passing of current from one converter to equipment assigned to its peers in case of system overload or failure).

Again, we extract an oracle that is composed of 23 trains (and its revisions over time) where each product model is composed of more than 1200 elements on average. They provided us with documentation of 56 bug reports, the approved reconfiguration sequences that triggers the bugs and the model fragments that contain the bugs. For each of the 56 bugs, we created a test case that includes the initial model, the set of reconfiguration rules and the model fragment where that bug was manifested and a bug description, all obtained from the documentation.

For this case study, we executed 30 independent runs for each of the 56 test cases for each approach (as suggested by [4]), i.e., 56 (bugs) * 3 (approaches) * 30 repetitions = 5040 independent runs.

5.4 Results

In this section, we present the results obtained for each case study in EBRo, the baseline and the RS. Figure 8 shows the charts with the recall and precision results for the industrial domain of BSH (left column) and CAF (right column). A dot in a graph represents the average result of precision and recall for each bug (37 bugs in BSH and 56 bugs in CAF) for the 30 repetitions. The first row of charts shows the results of EBRo, the second row of charts shows the results of the baseline approach, and the third row of charts shows the results of the RS approach.

Table 1: Mean values and standard deviations for precision, recall, F-Measure and MCC for each approach and each case study

		EBRo	Baseline	RS
BSH	Recall ± (σ)	83.40 ± 14.28	72.51 ± 14.07	35.19 ± 4.30
	Precision ± (σ)	78.20 ± 9.41	63.34 ± 9.95	40.22 ± 6.52
	F-measure ± (σ)	79.96 ± 9.33	66.43 ± 7.47	37.05 ± 3.07
	MCC ± (σ)	0.78 ± 0.10	0.63 ± 0.08	0.31 ± 0.04
CAF	Recall ± (σ)	80.12 ± 8.22	70.94 ± 9.52	41.87 ± 5.66
	Precision ± (σ)	76.35 ± 12.14	65.55 ± 11.26	38.81 ± 6.64
	F-measure ± (σ)	77.49 ± 7.81	67.26 ± 7.37	39.79 ± 4.44
	MCC ± (σ)	0.75 ± 0.08	0.64 ± 0.08	0.33 ± 0.05

Table 1 shows the mean values of recall, precision, F-measure, and MCC for the EBRo, baseline, and RS approaches in each case study. The EBRo approach obtains the best results in recall, precision, and MCC, providing an average value of 83.40% in BSH and 80.12% in CAF in recall, 78.20% in BSH and 76.35% in CAF in precision, and 0.78 in BSH and 0.75 in CAF in MCC. The second best results are obtained by the baseline, providing an average value of 72.51% in BSH and 70.94% in CAF in recall, 63.34% in BSH and 65.55% in CAF in precision, and 0.63 in BSH and 0.64 in CAF in MCC. The RS approach provided an average value of 35.19% in BSH and 41.87% in CAF in recall, 40.22% in BSH and 38.81% in CAF in precision, and 0.31 in BSH and 0.33 in CAF in MCC. In terms of recall, precision, and MCC, EBRo outperforms the rest of the approaches.

5.5 Statistical Analysis

Statistically significant differences can be obtained even if they are so small as to be of no practical value [4]. The goals of our statistical analysis are: (1) to provide formal and quantitative evidence (statistical significance) that EBRo does in fact have an impact on the comparison metrics (i.e., that the differences in the results were not obtained by mere chance); and (2) to show that those differences are significant in practice (effect size).

5.5.1 Statistical significance. Since our data does not follow a normal distribution in general, our analysis requires the use of non-parametric techniques. The Quade test shows that it is more powerful than the others when working with real data [15]. This tests provide a probability value, $p - Value$. The $p - Value$ obtains values between 0 and 1. It is accepted by the research community that a $p - Value$ under 0.05 is statistically significant [4]. The $p - Values$ obtained in the test are well below 0.05. Consequently, we can state that there are significant differences among the algorithms for the four performance indicators.

In addition, we perform a post hoc analysis. This kind of analysis performs a pair-wise comparison among the results of each algorithm, determining whether statistically significant differences exist among the results of a specific pair of algorithms. Specifically, we apply the Holm's Post Hoc analysis, as suggested by Garcia et al. [15].

Table 2: The $p - Values$ of Holm's post hoc analysis for each pair of algorithms and each case study

		EBRo vs Baseline	EBRo vs RS	Baseline vs RS
BSH	Recall	0.022	$1.1x10^{-14}$	$1.7x10^{-10}$
	Precision	$2.9x10^{-7}$	$\ll 2.2x10^{-16}$	$5.2x10^{-10}$
	MCC	$2.9x10^{-8}$	$\ll 2.2x10^{-16}$	$3.0x10^{-10}$
CAF	Recall	$3.8x10^{-5}$	$\ll 2.2x10^{-16}$	$9.2x10^{-15}$
	Precision	$3.0x10^{-5}$	$\ll 2.2x10^{-16}$	$1.2x10^{-14}$
	MCC	$1.4x10^{-10}$	$\ll 2.2x10^{-16}$	$1.4x10^{-14}$

Table 2 shows the $p - Values$ of Holm's post hoc analysis. All of the $p - Values$ shown in this table are smaller than their corresponding significance threshold value (0.05), indicating that the differences in performance between the algorithms are significant.

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

Table 3 shows the values of the size effect statistics. The largest differences were obtained between EBRo and the RS approach, where EBRo achieves better results all of the times. When comparing EBRo and the baseline, the differences are not so large, with EBRo achieving better results in around 80% of the times. EBRo obtained the best performance results among the three evaluated approaches (see Table 1).

The performed statistical analysis indicated that EBRo outperforms the rest of the approaches in terms of recall, precision, and MCC. Overall, these results confirm that the use of EBRo against the baseline and the RS approaches has an actual impact.

Table 3: The $\hat{A}_{12}$ statistic for each pair of algorithms and each case study

		EBRo vs Baseline	EBRo vs RS	Baseline vs RS
BSH	Recall	0.7166	1	1
	Precision	0.8524	1	0.9912
	MCC	0.8703	1	1
CAF	Recall	0.7548	1	1
	Precision	0.7411	1	0.9825
	MCC	0.8324	1	0.9992

5.6 Discussion

Our results confirm that the EBRo and the baseline approaches are better than random search based on the four metrics (recall, precision, F-measure, and MCC) on both the BSH and CAF case studies. Through this study, we concluded that there is empirical evidence to support the significance of the results of our algorithm. Thus, an intelligent algorithm is required to find good solutions to perform bug localization in reconfigurations of models at runtime.

In addition, the solutions obtained with the EBRo approach are better than the ones obtained with the baseline approach. The search space with the EBRo approach is driven by the reconfigurations, as it happens at runtime. While the baseline explores a much larger search space, any model conforms to the metamodel, and does not take into account the runtime reconfigurations.

However, the EBRo and baseline approaches are complementary. In the real world, before locating a bug, in most cases we do not know if the reconfigurations are the source of the bug for sure. Faced with a new bug, our recommendation is to start searching with EBRo, since EBRo not only obtains the configuration of the model where the bug occurs, but also provides the sequence of reconfigurations that leads to that particular configuration. In case the results obtained by EBRo are not useful, the baseline can be launched, since it explores a larger solution space than EBRo.

We realized that none of the approaches obtain a perfect solution. One of the issues that we detected that cause this outcome is the vocabulary mismatch. That means that for a specific concept, the terms used in the bug description are different from the terms used in the models. Nevertheless, this issue could be solved by augmenting the Natural Language Processing (NLP) with a dictionary of synonyms. In the same way, we have also detected cases in which in-house terms are used. Therefore, the regular dictionary of synonyms would not work in this case. This suggests that the dictionary of synonyms should be refined by domain engineers to include in-house terms. Another issue occurs when a bug description is incomplete. Omitting words in the bug descriptions negatively influences the fitness value of textual similarity since the fitness value of textual similarity is based on the co-occurrence of terms. This suggests that we must make the engineers aware of this issue. They should know that in cases in which the results obtained do not have enough quality, they can reformulate the descriptions of the bugs making the implicit knowledge explicit.

In addition, some solutions are invalid sequences of reconfigurations that, theoretically, are not going to take place since the necessary context changes can never occur. In our future work, we will study the introduction of crossover and mutation operators that, by construction, do not generate invalid sequences. We will also study a repair operation for invalid sequences. Although we are interested in the influence of narrowing the search space, we believe that we must maintain the possibility of generating sequences of theoretically invalid reconfigurations. For instance, induction hobs are sold all over the world and sometimes they are used in unforeseen ways, causing extremely unlikely sequences of reconfigurations to end up happening. These otherwise impossible reconfigurations are potential sources of bugs.

Results suggest that there is a relationship between the use of models at runtime and the quality of the solutions. The results are better the more you have specified the reconfigurations using the models at runtime. That is, the fewer flow control operators (e.g., ifs) and auxiliary variables are used to reconfigure, the better. This finding suggests that the models at runtime pay off when doing bug localization. However, the finding must be taken cautiously, since our evaluation was not designed to verify the influence of models at runtime on the quality of the solutions, but to compare the performance of different approaches in the location of bugs in the reconfigurations of models at runtime.

Finally, we have applied our EBRo approach to two real industrial cases, BSH and CAF, which are from very different domains. In both scenarios we have obtained quality results. Hence, the findings suggest that EBRo does not rely on the domain, since results depend on the reconfigurations themselves.

5.7 Threats to Validity

In this section, we present some of the threats to validity. We follow the guidelines suggested by De Oliveira et. al [11] to identify those that are applicable to this work.

Conclusion validity threats: We have identified three threats of this type. To address the *not accounting for random variation* threat, we considered 30 independent runs for each bug with each algorithm. In this paper we employed standard statistical analysis following accepted guidelines [4] to avoid the *lack of formal hypothesis and statistical tests* threat. To address the *lack of good descriptive analysis* threat, we have used precision, recall, F-measure and MCC metrics to analyze the confusion matrix obtained from the experiments; however, other metrics could be applied. In addition, some works argument that the use of the Vargha and Delaney A12 metric can be miss-representative [28] and that the data should be pre-transformed before applying them. We did not find any use case for data pre-transformation that applies to our case study.

Internal validity threats: We have identified two threats of this type. In this paper, we used standard values for the algorithms to avoid the *poor parameter settings* threat. These values have been tested in similar algorithms for feature localization [22]. In addition, the choice of the k value in the application of SVD can produce suboptimal accuracy when using LSI for software artifacts [29]. However, we plan to evaluate all of the parameters of our algorithm in a future work. The evaluation of this paper was applied to two industrial case studies from two of our partners, BSH and CAF, hence the *lack of real problem instances* threat is addressed.

Construct validity threats: We have identified one threat of this type. To address the *lack of assessing the validity of cost measures*

threat, we performed a fair comparison among EBRo, the baseline and the random approaches by generating the same number of reconfiguration sequences and using the same number of fitness evaluations.

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

6 RELATED WORK

In recent years, the research efforts in Search-Based Software Engineering (SBSE) in models at runtime are increasing.

McKinley et al. [26] applied SBSE to address uncertainty in adaptive systems. They integrated evolutionary computation into the development and runtime support of high-assurance, self-adaptive software. Their approach starts with requirements and moves through reconfigurable designs at runtime. Their work has been validated using experiments conducted in the context of robotics.

Andrade and de A Macêdo [1] have proposed a search-based approach for domain-independent representation of design spaces. Their approach (DuSE-MT) capture the most prominent design dimensions, their associated variation points, and the architecture changes requires to realize each solution. Their solutions are evaluated in terms of four quality metrics related to self-adaptation.

Williams et al. [39] presented a model-driven approach for adapting to dynamic runtime environments using metaheuristic optimization techniques. The metaheuristics exploit metamodels that capture the important components in the adaptation process. They contextualize the approach using an example and analyze different ways of applying the metaheuristic algorithms for discovering an optimal model of the case study's environment.

All of the above works present approaches that combine SBSE and models at runtime. However, these research efforts are focused on finding the optimal reconfigurations. In contrast, our work is focused on localizing bugs appearing as the result of dynamic reconfigurations of the system due to context changes.

In addition, there are other SBSE approaches for feature localization in models that, although not designed to locate bugs, could potentially be applied to that extent.

Lopez-Herrejon et al. [22] evaluate three standard search-based techniques with three objective functions in order to calculate the relationships of a feature model. Their results are slightly better for hill climbing than for the evolutionary algorithm, but they are not statistically significant when the first two objective functions are applied.

Harman et al. [17] performed a survey on the topic of search-based software engineering applied to Software Product Lines (SPLs). They present an overview of recent articles that are classified according to themes such as configuration, testing, or architectural improvement. Lopez-Herrejon et al. [21] performed a preliminary systematic mapping study at the connection of search-based software engineering and SPL. They categorized the articles using a known framework for SPL development.

Font et al. [13, 14] propose two approaches that use evolutionary algorithms to locate features in a model. They introduce a genetic operator for mutation that can work with a single model fragment and a crossover operator that combines two different product models. The results show that the use of a genetic algorithm allows the approach to provide accurate location of features in spite of inaccurate information on the part of the user.

In contrast to our work, all these feature location works only take into account the design time models.

In our previous work [3], we analyzed the influence of several timespan weightings on bug localization in models. We evaluated four timespan weightings: the most recent model modifications, the oldest model modifications, the mean of the modification timespan of the modified model elements, and the sum of the modification timespan of the modified model elements. The results showed that the use of the most recent timespan model modifications provides the best results in bug localization. In contrast to our previous works, our approach now takes into account the reconfigurations of models at run-time that can trigger the bug. Our future work includes the introduction of timespan weightings to improve the bug localization in the reconfigurations of models at runtime.

7 CONCLUSIONS

Bug localization is a significant maintenance activity and in a system with models at runtime, the reconfigurations that the model undergoes at runtime due to context changes can also be a source of bugs. In this paper, we have proposed an approach for bug localization in the reconfigurations of models at runtime (EBRo). Our approach is based on search-based software engineering, iterating on the reconfigurations search space using an evolutionary algorithm.

We evaluated our EBRo approach in two industrial case studies, BSH (firmware of induction hobs) and CAF (control software of trains), and compared it with the approach that they are using for bug localization and with a random search as sanity check. We determined which approach produces the best results in terms of precision, recall, F-measure, and MCC. To do this, we applied the approaches in BSH and CAF.

Results show that the study of the reconfiguration of models at runtime pays off for bug localization. Specifically, our EBRo approach outperforms the baseline and the random search approaches in terms of precision, recall, and MCC, reaching average precision and recall values of about 80% in BSH and 77% in CAF. Furthermore, results also suggest that our approach can be applied in real world environments. Finally, the statistical analysis of the results provides evidence of their significance.

ACKNOWLEDGMENTS

This work has been partially supported by the Ministry of Economy and Competitiveness (MINECO) through the Spanish National R+D+i Plan and ERDF funds under the project Model-Driven Variability Extraction for Software Product Line Adoption (TIN2015-64397-R). We also thank ITEA3 15010 REVaMP2 Project.

REFERENCES

[1] S. S. Andrade and R. J. de A Macêdo. 2013. Toward Systematic Conveying of Architecture Design Knowledge for Self-Adaptive Systems. In *2013 IEEE 7th International Conference on Self-Adaptation and Self-Organizing Systems Workshops*. 23–24. DOI: http://dx.doi.org/10.1109/SASOW.2013.13

[2] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. 2012. CoMA: Conformance Monitoring of Java Programs by Abstract State Machines. In *Proceedings of the Second International Conference on Runtime Verification (RV'11)*. Springer-Verlag, 223–238. DOI: http://dx.doi.org/10.1007/978-3-642-29860-8_17

[3] Lorena Arcega, Jaime Font, Øystein Haugen, and Carlos Cetina. 2017. On the Influence of Modification Timespan Weightings in the Location of Bugs in Models. In *Proceedings of the 26th International Conference on Information Systems Development, ISD 2017, Larnaca, Cyprus, September 6-8, 2017.*

[4] Andrea Arcuri and Lionel Briand. 2014. A Hitchhiker's Guide to Statistical Tests for Assessing Randomized Algorithms in Software Engineering. *Softw. Test. Verif. Reliab.* 24, 3 (May 2014), 219–250. DOI: http://dx.doi.org/10.1002/stvr.1486

[5] James E. Baker. 1987. Reducing Bias and Inefficiency in the Selection Algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms on Genetic Algorithms and Their Application*. L. Erlbaum Associates Inc., 14–21. http://dl.acm.org/citation.cfm?id=42512.42515

[6] Nelly Bencomo, Robert France, Betty H. C. Cheng, and Uwe Aßmann (Eds.). 2014. *Models@run.time. Foundations, Applications, and Roadmaps*. Springer International Publishing.

[7] Gordon Blair, Nelly Bencomo, and Robert B. France. 2009. Models@ Run.Time. *Computer* 42, 10 (Oct. 2009), 22–27. DOI: http://dx.doi.org/10.1109/MC.2009.326

[8] Sabri Boughorbel, Fethi Jarray, and Mohammed El-Anbari. 2017. Optimal classifier for imbalanced data using Matthews Correlation Coefficient metric. *PLOS ONE* 12, 6 (06 2017), 1–17. DOI: http://dx.doi.org/10.1371/journal.pone.0177678

[9] Ilhem Boussaïd, Patrick Siarry, and Mohamed Ahmed-Nacer. 2017. A survey on search-based model-driven engineering. *Automated Software Engineering* 24, 2 (01 Jun 2017), 233–294. DOI: http://dx.doi.org/10.1007/s10515-017-0215-4

[10] Paulo Casanova, Bradley Schmerl, David Garlan, and Rui Abreu. 2011. Architecture-based Run-time Fault Diagnosis. In *Proceedings of the 5th European Conference on Software Architecture (ECSA'11)*. Springer-Verlag, 261–277. http://dl.acm.org/citation.cfm?id=2041790.2041827

[11] Márcio de Oliveira Barros and Arilo Cláudio Dias-Neto. 2011. 0006/2011-Threats to Validity in Search-based Software Engineering Empirical Studies. *RelaTe-DIA* 5, 1 (2011).

[12] Ilenia Epifani, Carlo Ghezzi, Raffaela Mirandola, and Giordano Tamburrelli. 2009. Model Evolution by Run-time Parameter Adaptation. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, 111–121. DOI: http://dx.doi.org/10.1109/ICSE.2009.5070513

[13] Jaime Font, Lorena Arcega, Øystein Haugen, and Carlos Cetina. 2016. Feature Location in model-based Software Product Lines through a Genetic Algorithm. In *15th International Conference on Software Reuse (ICSR 2016)*. Limassol, Cyprus.

[14] Jaime Font, Lorena Arcega, Øystein Haugen, and Carlos Cetina. 2016. Feature Location in Models Through a Genetic Algorithm Driven by Information Retrieval Techniques. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MODELS '16)*. ACM, 272–282. DOI: http://dx.doi.org/10.1145/2976767.2976789

[15] Salvador García, Alberto Fernández, Julián Luengo, and Francisco Herrera. 2010. Advanced nonparametric tests for multiple comparisons in the design of experiments in computational intelligence and data mining: Experimental analysis of power. *Information Sciences* 180, 10 (2010), 2044 – 2064. DOI: http://dx.doi.org/10.1016/j.ins.2009.12.010 Special Issue on Intelligent Distributed Information Systems.

[16] Carlo Ghezzi, Andrea Mocci, and Mario Sangiorgio. 2012. Runtime Monitoring of Functional Component Changes with Behavior Models. In *Proceedings of the 2011th International Conference on Models in Software Engineering (MODELS'11)*. Springer-Verlag, 152–166. DOI: http://dx.doi.org/10.1007/978-3-642-29645-1_17

[17] M. Harman, Y. Jia, J. Krinke, W. B. Langdon, J. Petke, and Y. Zhang. 2014. Search Based Software Engineering for Software Product Line Engineering: A Survey and Directions for Future Work. In *Proceedings of the 18th International Software Product Line Conference - Volume 1 (SPLC '14)*. ACM, 5–18. DOI: http://dx.doi.org/10.1145/2648511.2648513

[18] Thomas K Landauer, Peter W. Foltz, and Darrell Laham. 1998. An introduction to latent semantic analysis. *Discourse Processes* 25, 2-3 (1998), 259–284. DOI: http://dx.doi.org/10.1080/01638539809545028

[19] Meir M Lehman, JF Ramil, and Goel Kahen. 2001. *A paradigm for the behavioural modelling of software processes using system dynamics.* Technical Report. Imperial College of Science, Technology and Medicine, Department of Computing.

[20] Dapeng Liu, Andrian Marcus, Denys Poshyvanyk, and Vaclav Rajlich. 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)*. ACM, 234–243. DOI: http://dx.doi.org/10.1145/1321631.1321667

[21] Roberto E. Lopez-Herrejon, Javier Ferrer, Francisco Chicano, Lukas Linsbauer, Alexander Egyed, and Enrique Alba. 2014. A Hitchhiker's Guide to Search-Based Software Engineering for Software Product Lines. *CoRR* abs/1406.2823 (2014).

[22] Roberto E. Lopez-Herrejon, Lukas Linsbauer, José A. Galindo, José A. Parejo, David Benavides, Sergio Segura, and Alexander Egyed. 2015. An assessment of search-based techniques for reverse engineering feature models. *Journal of Systems and Software* 103 (2015), 353 – 369. DOI: http://dx.doi.org/10.1016/j.jss.2014.10.037

[23] Usman Mansoor, Marouane Kessentini, Philip Langer, Manuel Wimmer, Slim Bechikh, and Kalyanmoy Deb. 2015. MOMM: Multi-objective model merging. *Journal of Systems and Software* 103 (2015), 423 – 439. DOI: http://dx.doi.org/https://doi.org/10.1016/j.jss.2014.11.043

[24] Shahar Maoz and David Harel. 2011. On tracing reactive systems. *Software & Systems Modeling* 10, 4 (01 Oct 2011), 447–468. DOI: http://dx.doi.org/10.1007/s10270-010-0151-2

[25] A. Marcus, A. Sergeyev, V. Rajlich, and J.I. Maletic. 2004. An information retrieval approach to concept location in source code. In *Proceedings of the 11th Working Conference on Reverse Engineering*. 214–223. DOI: http://dx.doi.org/10.1109/WCRE.2004.10

[26] Philip K. McKinley, Betty H. C. Cheng, Andres J. Ramirez, and Adam C. Jensen. 2012. Applying evolutionary computation to mitigate uncertainty in dynamically-adaptive, high-assurance middleware. *Journal of Internet Services and Applications* 3, 1 (01 May 2012), 51–58. DOI: http://dx.doi.org/10.1007/s13174-011-0049-4

[27] O. Moser, F. Rosenberg, and S. Dustdar. 2012. Domain-Specific Service Selection for Composite Services. *IEEE Transactions on Software Engineering* 38, 4 (July 2012), 828–843. DOI: http://dx.doi.org/10.1109/TSE.2011.43

[28] Geoffrey Neumann, Mark Harman, and Simon Poulding. 2015. *Transformed Vargha-Delaney Effect Size.* Springer International Publishing, 318–324. http://dx.doi.org/10.1007/978-3-319-22183-0_29

[29] A. Panichella, B. Dit, R. Oliveto, M. D. Penta, D. Poshyvanyk, and A. D. Lucia. 2016. Parameterizing and Assembling IR-Based Solutions for SE Tasks Using Genetic Algorithms. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 314–325. DOI: http://dx.doi.org/10.1109/SANER.2016.97

[30] Denys Poshyvanyk, Yann-Gael Gueheneuc, Andrian Marcus, Giuliano Antoniol, and Vaclav Rajlich. 2007. Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval. *IEEE Transactions on Software Engineering* 33, 6 (June 2007), 420–432. DOI: http://dx.doi.org/10.1109/TSE.2007.1016

[31] D. M. W. Powers. 2011. Evaluation: From precision, recall and f-measure to roc., informedness, markedness & correlation. *Journal of Machine Learning Technologies* 2, 1 (2011), 37–63.

[32] M. Revelle, B. Dit, and D. Poshyvanyk. 2010. Using Data Fusion and Web Mining to Support Feature Location in Software. In *IEEE 18th International Conference on Program Comprehension (ICPC)*. 14–23. DOI: http://dx.doi.org/10.1109/ICPC.2010.10

[33] Gerard Salton and Michael J. McGill. 1986. *Introduction to Modern Information Retrieval.* McGraw-Hill, Inc., New York, NY, USA.

[34] Daniel Schneider and Mario Trapp. 2013. Conditional Safety Certification of Open Adaptive Systems. *ACM Trans. Auton. Adapt. Syst.* 8, 2, Article 8 (July 2013), 20 pages. DOI: http://dx.doi.org/10.1145/2491465.2491467

[35] Hui Song, Michael Gallagher, and Siobhán Clarke. 2012. Rapid GUI Development on Legacy Systems: A Runtime Model-based Solution. In *Proceedings of the 7th Workshop on Models@Run.Time (MRT '12)*. ACM, 25–30. DOI: http://dx.doi.org/10.1145/2422518.2422523

[36] Michael Szvetits and Uwe Zdun. 2016. Systematic literature review of the objectives, techniques, kinds, and architectures of models at runtime. *Software & Systems Modeling* 15, 1 (01 Feb 2016), 31–69. DOI: http://dx.doi.org/10.1007/s10270-013-0394-9

[37] András Vargha and Harold D. Delaney. 2000. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132. DOI: http://dx.doi.org/10.3102/10769986025002101

[38] Thomas Vogel and Holger Giese. 2010. Adaptation and Abstract Runtime Models. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '10)*. ACM, 39–48. DOI: http://dx.doi.org/10.1145/1808984.1808989

[39] James R. Williams, Simon M. Poulding, Richard F. Paige, and Fiona Polack. 2013. *Exploring the Use of Metaheuristic Search to Infer Models of Dynamic System Behaviour.* 76–88.

[40] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 42, 8 (Aug 2016), 707–740.

13.3 SOSYM’19 Paper

Title:	An approach for bug localization in models using two levels: model and metamodel.
Authors:	Lorena Arcega, Jaime Font, Øystein Haugen, Carlos Cetina.
Journal:	Software & Systems Modeling.
Date:	March 2019
DOI:	`https://doi.org/10.1007/s10270-019-00727-y`
Contribution:	Lorena Arcega is the main author of the paper and is responsible for 90% of the work.

Software & Systems Modeling
https://doi.org/10.1007/s10270-019-00727-y

REGULAR PAPER

An approach for bug localization in models using two levels: model and metamodel

Lorena Arcega[1,2] · Jaime Font[1,2] · Øystein Haugen[3] · Carlos Cetina[1]

Received: 16 March 2018 / Revised: 24 January 2019 / Accepted: 4 March 2019
© Springer-Verlag GmbH Germany, part of Springer Nature 2019

Abstract
Bug localization is a common task in software engineering, especially when maintaining and evolving software products. This paper introduces a bug localization approach that, in contrast to existing source code approaches, takes advantage of domain information found in the model and the metamodel. Throughout this paper, we present an approach for bug localization in models (BLiM2) that applies the source code ideas for bug localization (textual similarity to the bug description and the Defect Localization Principle) and takes advantage of the domain information from the model and the metamodel. We evaluated our approach in BSH, a real-world industrial case study in the induction hob domain measuring the results in terms of recall, precision, the combination of both the F-measure and the Matthews correlation coefficient. Our study shows that our BLiM2 approach, which combines information from the model and the metamodel for the textual similarity and differentiates between the timespan from the model and metamodel, provides the best results in this work. We also performed a statistical analysis to provide evidence of the significance of the results. The values obtained show that there exist significant differences in the performance of the best BLiM2 approach with the approach used by our industrial partner. Finally, the effect size statistics reveals that the best BLiM2 approach obtains better results in the 78% of the times in the worst case.

Keywords Bug localization · Model-driven engineering · Reverse engineering

1 Introduction

Nowadays, software exists in almost everything. This trend has been accompanied by a high increase in the scale and the complexity of software. Unfortunately, this is also accompanied by more software bugs. Hence, software maintenance is becoming increasingly important. Lehman et al. [33] 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.

Bug localization is one of the most important and common activities performed by developers during software maintenance and evolution. Bug localization aims to identify the location in the artifact that is pertinent to a software fault. A recent survey [61] reveals that none of the bug localization approaches take into account models as the source of the bugs. In the model paradigm, models can play several roles in software development: diagrams for analysis can be reverse-engineered from source code, or can be used for code generation. In this work, we focus on models for code generation. When models are used for code generation, addressing bugs at the model level must not be neglected.

In the case of bug localization in source code, approaches are based mainly on information retrieval [50,65]. Some works [54,62] also take into account the Defect Localization Principle. This principle is based on the observation that the most recent modifications to a project are most likely

Communicated by Prof. Lionel Briand.

✉ Lorena Arcega
larcega@usj.es

Jaime Font
jfont@usj.es

Øystein Haugen
oystein.haugen@hiof.no

Carlos Cetina
ccetina@usj.es

1 Escuela de Arquitectura y Tecnologia, Universidad San Jorge, Zaragoza, Spain

2 Department of Informatics, University of Oslo, Oslo, Norway

3 Faculty of Computer Science, Østfold University College, Halden, Norway

Published online: 14 March 2019

the cause of future bugs [23,66]. Taking into account recent modification may lead to finding relevant locations that are the cause of a bug [54].

In this paper, we propose an approach for bug localization in models. This approach enables us to evaluate to what extent the ideas that have been successfully applied in source code for bug localization (textual similarity to the bug description and the Defect Localization Principle) also work for models. Our approach takes into account a property of the models that is not present in source code, namely the domain information embedded in models and metamodels.

In our approach, information retrieval is used to measure the similarity of model fragments with the description of the bug report that we want to locate. Model fragments are formed by model elements, and each model element has an associated modification time. The Defect Localization Principle is measured through the timespan weighting that assesses the model fragment solutions using the timespan of the model modifications.

We materialize our bug localization approach as a multi-objective evolutionary algorithm that uses both the similarity to the bug report and the timespan weighting as fitness functions. Our approach is, in fact, a family of approaches. We can change from one of our approach to another by changing how we use the information of the models and the metamodels with which we measure the similarity and the timespan. As a result, software engineers obtain a ranked list of model fragments from the model, which is intended to identify the parts of the model that are relevant to the bug.

We have applied our approach to the product models from BSH, one of the largest manufacturers of home appliances in Europe. Its induction division has been producing induction hobs under the brands of Bosch and Siemens for the last 15 years. The firmware that controls the induction hobs is specified by means of a domain-specific language (IHDSL). The different configurations of the induction hobs are managed following a model-based software product line (SPL) approach that uses common variability language (CVL) [24] to configure their models. The firmware of their products is generated from the IHDSL models.

In our evaluation, we compare our bug localization approach, which uses model and metamodel information (BLiM2), with a baseline approach. The baseline approach is used by BSH for bug localization. We apply both the BLiM2 approach and the baseline to the product family of BSH. They provided us with documentation about bugs. For each one of the bugs, the documentation provided a bug description and the localization of the bug. Taking the bug descriptions and the product family as input, we measure the results using the standard measurements accepted by the software engineering community: recall, precision, the combination of both the F-measure, and the Matthews correlation coefficient (MCC) [40,51] using the location of the bugs as oracle.

The variant BLiM2-3OT of our approach achieves the best results. It has three objectives in the fitness function: one that combines information from the model and the metamodel for text similarity; one that takes into account the modification timespan of the model; and one that takes into account the modification timespan of the metamodel. The results in terms of recall, precision, and MCC, on average are 90.30%, 79.66%, and 0.83, respectively. Finally, we perform a statistical analysis on the results in order to provide quantitative evidence of the impact of both the BLiM2 approach and the baseline approach and to show that this impact is significant.

The remainder of the paper is structured as follows. In Sect. 2, we present the domain-specific language used by our industrial partner and the differences between feature localization and bug localization. In Sect. 3, we describe our BLiM2 approach. In Sect. 4, we evaluate the application of our approach in BSH. In Sect. 5, we examine the related work of the area. Finally, we present our conclusions in Sect. 6.

2 Background

The running example and the evaluation in this paper are performed using the products of our industrial partner, BSH. In this section, we present the domain-specific language (DSL) that is used by BSH to formalize their products, which is called IHDSL. The common variability language (CVL) is also presented. CVL is the language used by our approach to formalize the model fragments.

The newest induction hobs (IHs) feature full cooking surfaces, where dynamic heating areas are automatically generated and activated or deactivated depending on the shape, size, and position of the cookware placed on the top. In addition, there has been an increase in the type of feedback provided to the user while cooking. All of these changes have been made possible at the expense of increasing the complexity of the software behind IHs.

The domain-specific language used by our industrial partner to specify the induction hobs (IHDSL) is composed of 46 meta-classes, 47 references among them, and more than 180 properties. To gain legibility and due to intellectual property right concerns, in this section, we show a simplified subset of the IHDSL (see Fig. 1, IHDSL metamodel and IHDSL syntax). However, the evaluation was performed using the full IHDSL that is used in BSH.

Inverters are in charge of transforming the input electric supply to match the specific requirements of the IH. Then, the energy is transferred to the inductors through the channels. There can be several alternative channels, which enable different heating strategies depending on the cookware placed on top of the IH at runtime. The path followed by the energy through the channels is controlled by the power manager.

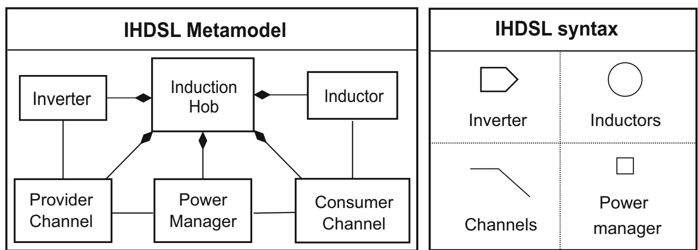


Fig. 1 IHDSL metamodel

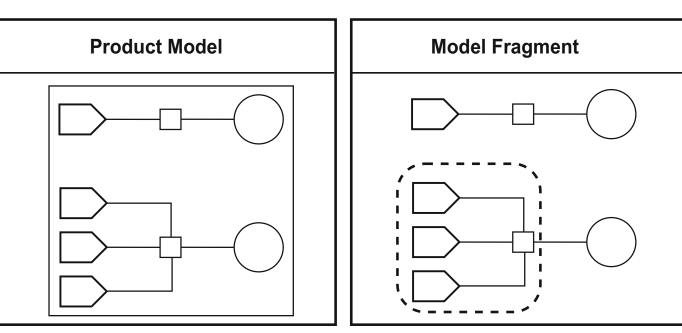


Fig. 2 IHDSL syntax, product model, and model fragment formalization

Inductors are the elements where the energy is transformed into an electromagnetic field.

The product model in Fig. 2 depicts an example of a product model that is specified with the IHDSL. The product model contains four inverters that are used to power two different inductors. The upper inductor is powered by a single inverter, while the lower inductor is powered by the combination of three different inverters. Power managers act as hubs to perform the connection between the inverters and the inductors.

To formalize the solution of our approach as model fragments, we use common variability language (CVL) [24,55], due to its capabilities to formalize a set of model elements as a model fragment. The model fragment in Fig. 2 shows an example of a model fragment of the product model (the product model in Fig. 2). The model fragment includes the three inverters (in charge of powering the lower inductor), the three channels, and the power manager that is used to aggregate and manage the power provided by those inverters. Then, the solution of our approach is formalized by means of CVL and shown to the engineers.

In addition, to address the evolution of the metamodel, our industrial partner uses the variable metamodel strategy (VMM) [17]. This strategy has achieved better results than the migration strategy in domains like that of our industrial partner, BSH, in terms of indirection, automation, and trust leak.

2.1 Differences between feature and bug localization

In software systems, a feature represents a functionality that is defined by requirements and is accessible to developers and users. Software maintenance and evolution involve adding new features to programs, improving existing functionalities, and removing bugs, which is analogous to removing unwanted functionalities [14].

Bug localization is a specialization of feature localization. In the end, the software engineer obtains a piece of software that is in charge of some functionality in the system with either of the approaches. However, if we want to perform bug localization, we must take into account different particularities than if we want to perform feature localization. For example, the Defect Localization Principle can be applied to bug localization but not to feature localization.

Our previous approach [19] was developed for feature localization. It uses an evolutionary algorithm that iterates through the models of the system and assesses each one of the possible solutions. In the end, the software engineer obtains an ordered set of model fragments that fits the feature.

In this work, we adapt our feature localization approach [19] to obtain better solutions in bug localization. We use the same operations to iterate through the models; however, this approach has a new fitness function. The feature localization approach assesses each model fragment regarding textual

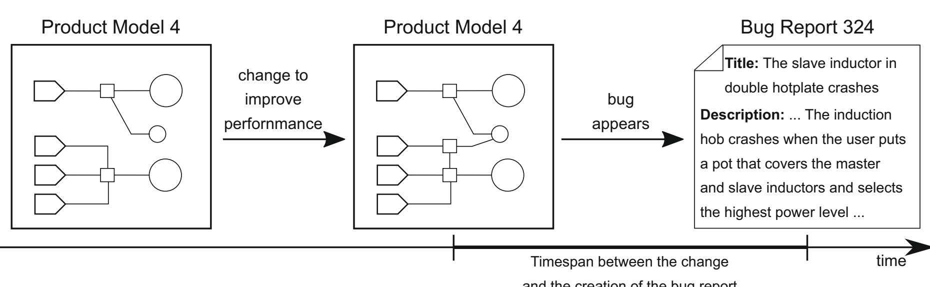


Fig. 3 Example of a bug

similarity with the feature description. In addition to the textual similarity, this approach adds a fitness function that is specific for use with bugs. This fitness function exploits the timespan since the last modification.

Another difference is that our BLiM2 approach takes into account a property that is specific to the models. It uses the domain information embedded in the model and the metamodel. In other words, to assess each model fragment, our approach takes information from both model and metamodel (text for measure textual similarity and timespan for measure the last modification). Nevertheless, the use of the domain information embedded in the metamodel is not specific for bug localization, it could be applied to feature localization.

In this way, we convert the generic feature localization approach into a bug localization approach. This approach is a specialization of our previous feature localization approach. With our previous feature localization approach, users can locate features and bugs. In fact, the software engineers of BSH use our feature localization approach to locate bugs. The new specific bug localization approach exploits the timespan dimension of the last modifications, which is only relevant in bug localization.

3 The BLiM2 approach

Figure 3 shows an example of the use of our approach. The product model that appears on the left of the timeline is modified to make an improvement in the performance of its inductors. As a result, another product model is generated. This product model has a new power manager that connects the small inductor with the rest of the inverters. After some time of use, a bug appears and a bug report is generated. In the reminder of this paper, the timespan between the change and the creation of the bug report is called the modification timespan.

We materialize our BLiM2 approach as a multi-objective evolutionary algorithm that uses both the similarity to the bug description and the modification timespan. The use of a multi-objective algorithm allows to show the results of both objectives (similarity and modification timespan). The effectiveness of each objective is different for each bug localization. Sometimes the similarity will be more successful to find the model fragment that contains the bug and sometimes the timespan is the more successful.

The objective of bug localization in models is to obtain a set of model fragments from a given set of models that may correspond to a specific bug description being provided by the user of the approach. To do this, the approach receives a set of models and relies on an evolutionary algorithm that iterates a set of model fragments and evolves them using genetic operations. The evolutionary algorithm is guided by a fitness operation. As output, the approach provides a list of model fragments that should contain the bug. The overview of the process is shown in Fig. 4.

The left part of Fig. 4 shows the inputs used in our approach. The input is composed of a set of models, a metamodel, and a bug description:

- A set of product models that contain the target bug. The software engineer selects a set of product models from the entire family of products that contain the bug to be located.
- A metamodel to which the models of the product family conform.
- A bug description of the target bug, using natural language. Typically, these descriptions come from textual documentation of a bug report. Therefore, the query will include some domain-specific terms that are similar to those used when specifying the product models. The knowledge of the engineers about the domain and the product models will be useful for selecting the description from the bug report.

The right part of Fig. 4 shows the main steps of our approach.

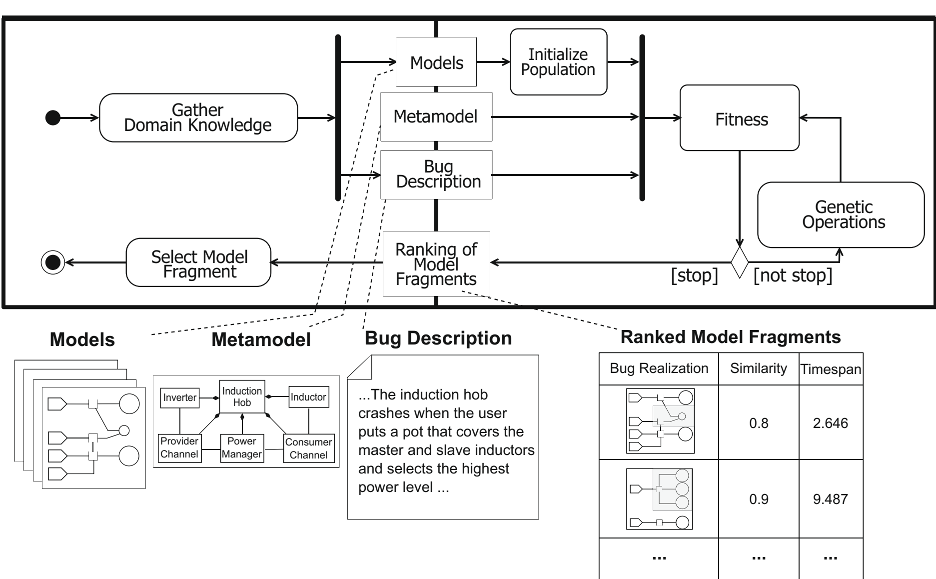


Fig. 4 Overview of the bug localization approach in models: BLiM2

- The **initialize population** step calculates an initial set of model fragments from the input set of product models. This initial set of model fragments is randomly extracted from the product models. This is a common practice in evolutionary computation; as an alternative, seeds (fragments of a model chosen manually) can be proportionated in order to optimize the population, although that is out of the scope of this work.
- The **genetic operations** generate the new generation of model fragments. First, a selection operation selects the model fragments that will be used as parents of the new model fragments. The fitness values (see Sect. 3.2) are used to ensure that the best model fragments are chosen as parents. Then, a crossover operation mixes the model elements of the two parents into a new model fragment. Finally, a mutation operation introduces variations in the new model fragment (by adding or removing model elements) in hopes that the new model fragment achieves better fitness values than its parents.
- The **fitness** step assigns values that assess how good each model fragment is. The values will take into account the following terms:
  - Bug description: The more terms shared between the bug description and the properties of the model fragment (from the model and the metamodel), the higher this fitness value.
  - Modification timespan: The more recent the modification time, the higher this fitness value is. The timespan is calculated based on the difference between the last modification of a model element (in the model or in the metamodel) and the day on which the bug was discovered (see Fig. 3).

The output of BLiM2 (see Fig. 4) is an ordered set of model fragments that contains the target bug. The software engineer obtains this set of model fragments, which is intended to identify the parts of the model that are relevant to the bug. To do so, the engineer can order the ranking following different criteria depending on the variant of the approach used, such as the similarity to the bug description, and the most recent model fragment modifications.

Overall, the aim of the approach is to find the most relevant model fragment that contains the bug described by the bug report. To do so, the approach performs a search guided by a fitness function. This search is done among the different model fragments (previously obtained applying mutation and crossover operations) that could contain the bug description. The fitness function will assign values based on the similarity with the textual description and the most recent model modifications of that model fragment.

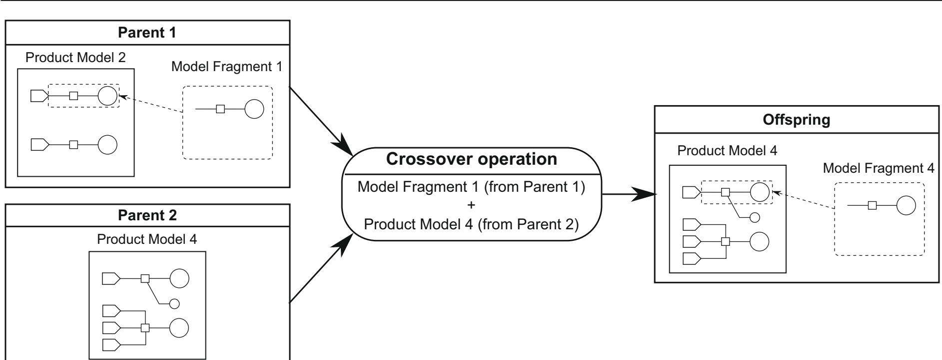


Fig. 5 Crossover operation

The following sections describe the genetic operations of BLiM2 to generate new model fragments and how the fitness of each model fragment is determined in terms of similarity to the bug description, and the time of model modifications.

3.1 Genetic operations of the BLiM2 approach

The generation of subsequent model fragment sets is performed by applying genetic operators that we have adapted to work on model fragments in a previous work [19]. In other words, new fragments based on the existing ones are generated through the use of two genetic operators: the crossover operator and the mutation operator. These two operations are summarized in the following subsections, but the details are in [19].

3.1.1 Crossover

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

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

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

3.1.2 Mutation

Figure 6 shows an example of our mutation for model fragments. Each model fragment is associated with a product model, and the model fragment mutates in the context of its associated product model. In other words, the model fragment will gain or drop some elements, but the resulting model fragment will still be part of the referenced product model. The mutation possibilities of a given model fragment are driven by its associated product model.

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

- **Subtractive mutation**: This kind of mutation randomly removes some elements from the model fragment. The only constraint is that the elements be selected from the edges of the model fragment (they are connected with a single element). Therefore, the resulting model fragment is still connected (we can navigate from any element to any other element through the connections between the elements), and it is not split into several isolated groups of elements. Since the resulting model fragment is a sub-

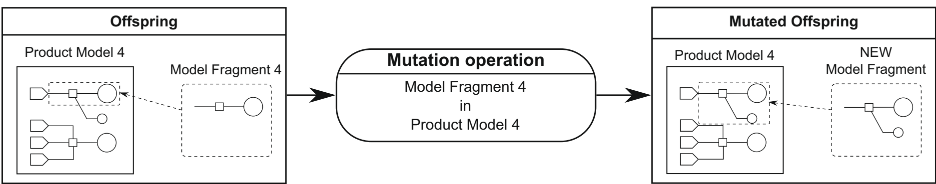


Fig. 6 Mutation operation

set of the original model fragment and the original was present in the referenced product model, the resulting product model will always be present in the referenced product model.

– **Additive mutation**: This kind of mutation randomly adds some elements to the model fragment. The only constraint is that the resulting model fragment be part of the referenced product model. To achieve this, the boundaries of the model fragment with the rest of the product model are identified and then a random element from the boundary is added to the resulting model fragment. By doing so, the mutated model fragment will be part of the referenced product model.

Both crossover and mutation operations are designed to produce individuals, by selecting subset of existing models to conform the new model fragment. The focus of this paper is generating a subset of model elements that are relevant to the bug that the domain expert wants to locate. The resultant model fragments are then presented to a domain expert for his assessment of its relevance for finding the bugs.

As a result, a new model fragment is created, but it still references the same product model. In other words, the model fragment represents another possible solution that can contain the bug (that is present in the product model) for the specific bug being located.

3.2 Fitness of the BLiM2 approach

In evolutionary algorithms, the fitness step is used to assess the different degrees of adaptation to the environment that different model fragments have. Following this idea, our fitness step is used to determine the suitability of each model fragment to the problem. The input of this step is a set of model fragments, and the output produced is a set where each model fragment has been assigned with two fitness values: the similarity to the feature description and the timespan to the most recent model fragment modifications.

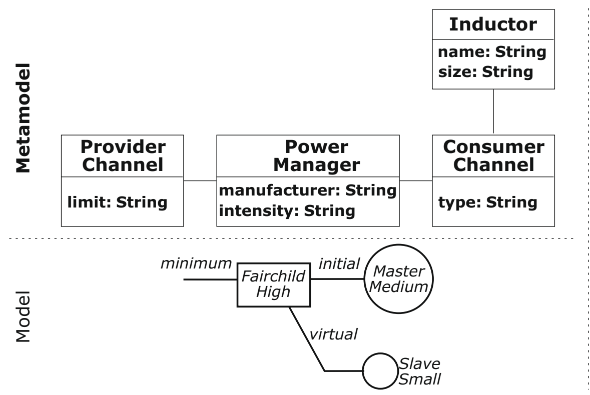
3.2.1 Model fragment similarity to the bug description

To assess the relevance of each model fragment in relation to the bug description provided by the user, we apply methods based on information retrieval (IR) techniques. There are many popular IR techniques such as vector space model (VSM) [52], latent semantic indexing (LSI) [13] or latent Dirichlet allocation (LDA) [8]. The research findings are ambiguous and contradictory about which technique provides the best performance [57].

Based on our previous experience [4,19], we apply latent semantic indexing (LSI) to analyze the relationships between the description of the bug provided by the user and the model fragments. Besides that LSI provides good results when applied to source code [35,45,49], a recent work reveals that LSI performs better when applied to text [47]. Product models are representations at a higher abstraction level than the source code, and the language used to build them is closer to the bug description language; similar to text.

LSI constructs vector representations of a query and a corpus of text documents by encoding them as a term by document co-occurrence matrix, (i.e., a matrix where each row corresponds to terms and each column corresponds to documents, with the last column corresponding to the query). Each cell holds the number of occurrences of a term (row) inside a document or the query (column). Figure 7 shows the term extraction from a model fragment. The text of the document corresponds to the names and values of the properties and methods of each model fragment from the model and the metamodel, see Fig. 7. The metamodel provides the names of classes, attributes, and methods, while the model provides the values of these attributes and methods. The left part of Fig. 7 shows a model fragment with its corresponding metamodel fragment. The right part of Fig. 7 shows the terms extracted, the number of occurrences, and the source of the term (which comes from the model or from the metamodel). For example, the term channel appears 2 times in the fragment and the term channel comes from the metamodel (see the second column of the table in Fig. 7).

In our work, the documents are model fragments, i.e., a document of text is generated from each of the model fragments. The query is constructed from the terms that appear in the bug description. If the textual terms used for the model and the bug description differ too much, the LSI will not work. Therefore, the text from the documents (model fragments) and the text from the query (bug description) are homogenized by applying well-known natural language pro-



Term	Occurrences	Source
provider	1	Metamodel
channel	2	Metamodel
type	1	Metamodel
limit	1	Metamodel
manager	1	Metamodel
intensity	1	Metamodel
size	1	Metamodel
name	1	Metamodel
master	1	Model
slave	1	Model
high	1	Model
...	...	...
small	1	Model

Fig. 7 Terms extraction from a model fragment and the corresponding metamodel

Fig. 8 LSI applied to a model fragment

Model Fragments

Terms	MF1	MF2	MF3	MF4	...	MFn	Query
size	6	12	9	6	...	12	6
provider	12	6	4	0	...	24	10
coil	36	30	0	0	...	0	0
small	24	12	2	0	...	6	4
intensity	0	8	10	2	...	0	6
high	6	0	2	0	...	6	8
...	...	...	...	...	...	...	0
name	8	2	1	12	...	3	0

cessing techniques (tokenizing, parts-of-speech tagging, and lemmatizing) to eventually reduce this gap. If the languages used differ too much, other techniques such as manual annotation of the model elements could be applied at the expense of increasing the effort.

The union of all the keywords extracted from the documents (model fragments) and from the query (bug description) is the terms (rows) used by our LSI fitness (see Fig. 8). Each column is one of the model fragments; for example, the column that is shaded in gray in Fig. 8 is the one that corresponds to the model fragment from Fig. 7. The last column is the query obtained from the bug description of the user. Each row is one of the terms extracted from the corpuses of text composed by all of the model fragments and the query itself. Each cell has the number of occurrences of each of the terms in the model fragments.

Once the matrix is built (see Fig. 7), we normalize and decompose it into a set of vectors using a matrix factorization technique called singular value decomposition (SVD) [31]. One vector that represents the latent semantics of the document is obtained for each model fragment and the query. Finally, the similarities between the query and each model fragment are calculated as the cosine between the two vectors. The fitness value that is given to each model fragment is the one that we obtain when we calculate the similarity, obtaining values between −1 and 1.

To make the variants of our approach, we can take the union of the terms from the model and the metamodel

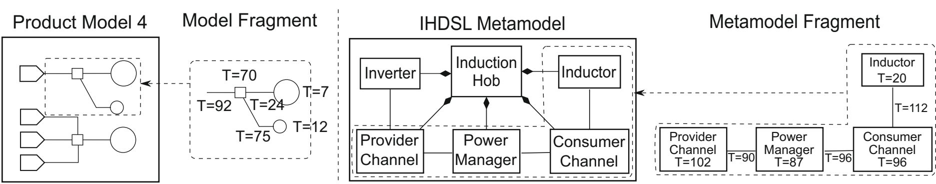


Fig. 9 Timespan of the modifications of the model elements of a fragment

(as shown in the example), or we can obtain different co-occurrence matrices: one co-occurrence matrix from the terms of the metamodel and another co-occurrence matrix from the terms of the model. This way, we will obtain two fitness values instead of one.

3.2.2 The most recent model modification

To apply the Defect Localization Principle, we measure the timespan to the last modifications of the model. As the modifications affect the model elements, each model element has its own modification timespan. Thus, each model element has a constant value of modification timespan during the execution of the algorithm, but different model elements have different modification timespan values.

The modifications of the model over time are considered when extracting the most relevant model from the target bug (the time difference between the last modification of a model element and the usage day). A recently modified model element (i.e., a short timespan) has a lower timespan value than another model element that was modified farther in the past. Since a model fragment is composed by a set of model elements, the timespan weighting of the model fragment depends on the timespan weightings of the model elements that compose it.

The time difference is based on the number of days and can therefore be very large when the model fragment was modified a long time ago. To normalize the time difference, mathematical solutions such as square root or logarithm can be used. We used square root because it has achieved good results in other works that use time differences [27,62]. The use of square root is more suitable and more effective for the proposed approach [66].

We consider the most recent model modification timespan as the modification timespan weighting of a model fragment. We chose this assessment because it achieved the best results in our previous work [4]. This function expresses the concern of capturing primarily the model fragments with the model elements that have the lowest modification timespans, i.e., model elements that have been recently modified. Then, the value of the model fragment will be the value of the most recently modified model element.

In this work, the modification timespan can come from the metamodel or from the model. If we follow the function presented in [4], in the example of Fig. 9, the value of the model fragment is 7 days, which means a square root of 2.646. However, we can obtain two fitness values, one from the model and another from the metamodel. The value of the model fragment will remain 7 days (a square root of 2.646), while the value of the metamodel fragment will be 20 days (a square root of 4.472). This way, we will obtain two fitness values instead of one. A variant of our BLiM2 applies these two fitness values.

3.3 BLiM2 variants

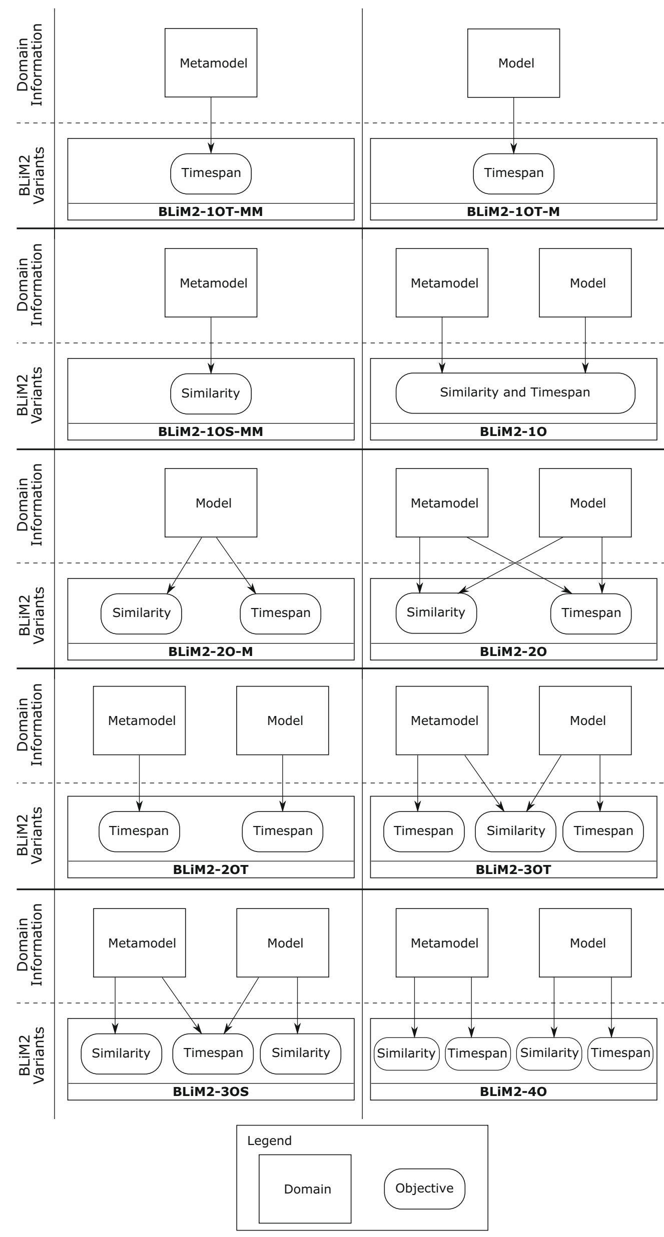
When considering the possible variants of the BLiM2 approach, we have taken into account the following facts, which are extracted from the previous subsections:

1. Information retrieval techniques are widely used in bug localization, specifically LSI, which provides good results applied to source code [35,45,49].
2. The Defect Localization Principle improves the results of the LSI when applied together [4,54,62].

Figure 10 shows the variants of BLiM2 taking into account these two facts. The figure shows the BLiM2 approach with the fitness function and the combinations of the information from the model and the metamodel, for a total of ten variants.

1. BLiM2 with one objective in the fitness function taking information from the metamodel in timespan (BLiM2-1OT-MM).
2. BLiM2 with one objective in the fitness function taking information from the model in timespan (BLiM2-1OT-M).
3. BLiM2 with one objective in the fitness function taking information from the metamodel in similarity (BLiM2-1OS-MM).

Fig. 10 Variants of the BLiM2 approach



4. BLiM2 with one objective in the fitness function getting two fitness values from information from the model and the metamodel in similarity and in timespan but combining these objectives in one with equal weight (BLiM2-1O).
5. BLiM2 with two objectives in the fitness function taking information from the model in similarity and timespan (BLiM2-2O-M).
6. BLiM2 with two objectives in the fitness function combining information from the model and the metamodel in similarity and timespan (BLiM2-2O).
7. BLiM2 with two objectives in the fitness function combining information from the model and the metamodel in timespan (BLiM2-2OT).
8. BLiM2 with three objectives in the fitness function combining information from the model and the metamodel in similarity and getting one value for each one in timespan (BLiM2-3OT).
9. BLiM2 with three objectives in the fitness function combining information from the model and the metamodel in timespan and getting one value for each one in similarity (BLiM2-3OS).
10. BLiM2 with four objectives in the fitness function getting two fitness values from information from the model and the metamodel in similarity and in timespan (BLiM2-4O).

3.4 Implementation details

Our algorithm is based on NSGA-II [12], one of the most frequently used multi-objective evolutionary algorithms. Given a set of model fragments where each model fragment has a fitness value for its bug similarity (see Sect. 3.2.1) and its recent time modifications (see Sect. 3.2.2), NSGA-II orders these model fragments by means of non-dominated sorting. A model fragment is non-dominated if the following holds: (1) there is no other model fragment that is better than the current one for some fitness value; and (2) the current model fragment does not worsen other fitness values. As a result, NSGA-II finds Pareto-optimal model fragments.

We implemented the multi-objective evolutionary algorithm as outlined in Algorithm 1. During the generation of the initial set (lines 3–8), model fragments are randomly generated. The algorithm evaluates the fitness for each model fragment and adds it to the initial set. During the evolution process (lines 11–18), new offspring are generated as a result of selecting model fragments (by means of the binary tournament selection), recombining them and applying a mutation operation. The algorithm evaluates the fitness for each offspring and adds it to a temporal set (line 17). By means of a combination of non-dominated sorting and crowding distance sorting [12], the algorithm selects the model fragments from both the old set and the temporal set (line 19) to create a new set.

Algorithm 1 BLiM2 (NSGA-II-based)

```
Require: Size, p_c, p_m, MaxEval
Ensure: PF ← set of nondominated solutions
 1: P = 0
 2: evaluations = 0
 3: for (i = 1 to Size) do
 4:    s ← NewSolution()
 5:    EvaluateFitness(s)
 6:    evaluations ← evaluations + 1
 7:    P ← P + 1
 8: end for
 9: while (evaluations < MaxEval) do
10:    P_O ← 0
11:    for (i = 1 to Size/2) do
12:       parents ← Selection(P)
13:       offspring ← Crossover(offspring)
14:       offspring ← Mutation(parents)
15:       EvaluateFitness(offspring)
16:       evaluations ← evaluations + 1
17:       P_O ← P_O + offspring
18:    end for
19:    P ← P ∪ P_O
20: end while
21: PF ← BestFrom(P)
22: RankingAndCrowdingDistance(P)
```

We performed some parameter tuning to find the best values for the parameters of our algorithm. However, we did not find large differences that had an effect in our main focus. The focus of this paper is not to tune the values to improve the performance of the algorithms when applied to a particular problem, but rather to compare the performance of the algorithms in terms of solution quality. Then, we have principally chosen values for those settings that are commonly used in the literature [53]. As suggested by Arcuri and Fraser [6], default values are good enough to measure the performance of search-based techniques in the context of testing. Hence, the crossover operation is applied with a crossover probability (p_c) of 0.9, and the mutation operation is applied with a probability (p_m) of 0.1.

The number of generations (repetitions of the genetic operations and fitness loop) allowed for the algorithm is 2500 since it is the value needed by our case study to converge (note that this value is case specific). The rest of the settings are detailed in Table 1. There are two atomic performance measures for evolutionary algorithms: one regarding solution quality and another regarding algorithm speed. In this work, we focus on the solution quality, determining the variant that provides solutions that are closer to the one from the oracle in terms of recall, precision, and MCC. Nevertheless, the time spent by each variant to reach the limit of 2500 generations is around 60 s.

We have used the Eclipse Modeling Framework to manipulate the models and CVL to manage the fragments of

Table 1 BLiM2 configuration parameters

Parameter description	Value
$Size$: population size	100
μ: number of parents	2
λ: number of offspring from μ parents	2
r: solutions replaced by set size	2
p_c: crossover probability	0.9
p_m: mutation probability	0.1

models. The IR techniques used to process the language have been implemented using OpenNLP [1] for the POSTagger and the English version of the Porter 2 [56] as the stemming algorithm. The LSI has been implemented using the Efficient Java Matrix Library (EJML [16]). The genetic operations are built upon the Watchmaker Framework for Evolutionary Computation [15].

4 Evaluation

This section presents the evaluation of our approach: the oracle preparation, the experimental setup, a description of the case study where we applied the evaluation, the results obtained, the statistical analysis performed, the discussion of the results, and the threats to validity.

To evaluate the approach, we applied it to an industrial case study from our industrial partner: BSH, a leading manufacturer of home appliances in Europe.

4.1 Oracle preparation

The oracle is the ground truth and is used to compare the results provided by the BLiM2-X approaches, the baseline, and the random search (RS) that works as sanity check. The baseline is the approach used in our industrial partner for bug localization [19]. As we explained in Sect. 2.1, a bug can be seen as an unwanted functionality, and a feature represents a functionality. For this reason, the feature localization approach presented in Font et al. [19] can be used for bug localization. Even though it was designed with a more general purpose in mind (feature localization), it is the best they have for bug localization in models.

To prepare the oracle, our industrial partner provided us with the bug reports that have occurred in the product models. These bug reports contain natural language bug descriptions and the approved model fragments that contain the target bugs.

4.2 Experimental setup

This experiment evaluates whether or not the information found in the metamodel improves the bug localization results. In addition, we compare the BLiM2-X approaches with the baseline [19] and with a random search (RS) sanity check. If RS outperforms an intelligent search method, we can conclude that there is no need to use metaheuristic search. We use the name BLiM2-X to represent any of the variants of our approach; the letter 'X' represents one of '1OT-MM,' '1OT-M,' '1OS-MM,' '1O,' '2O-M,' '2O,' '2OT,' '3OT,' '3OS,' and '4O.'

Figure 11 shows an overview of the process that was followed to evaluate our BLiM2-X approach. The left part of the figure shows the inputs of the evaluation process provided by our industrial partner, which are the product family and bug reports. The product family and bug descriptions are used to run BLiM2-X and the baseline approaches. We run each of the approaches and obtain a ranking of model fragments that we can compare with an oracle in order to check accuracy. The inputs of the evaluation process, which are the product family and bug reports, were provided by our industrial partner.

Therefore, in order to compare the baseline and the RS approaches with BLiM2-X, we first take the best solution of the baseline and RS approaches. Second, we take the best solution of each BLiM2-X. Finally, these solutions are then compared to the model fragment that contains the target bug of the oracle in order to get a confusion matrix.

A confusion matrix is a table that is often used to describe the performance of a classification model (in this case, BLiM2-X and the baseline) on a set of test data (the solutions) for which the true values are known (from the oracles). In our case, each solution that is output by the approaches is a model fragment composed of a subset of the model elements that are present in the product model (where the bug is being located). Since the granularity will be at the level of model elements, the presence or absence of each model element will be considered as a classification. Therefore, our confusion matrices will distinguish between two values (TRUE or presence and FALSE or absence).

Figure 12 shows an example of the comparison process that is performed to compare a result from one of the evaluated approaches with the ground truth from the oracle and the resulting confusion matrix. We obtain a confusion matrix for each of the solutions predicted by each of the approaches. The left part of Fig. 12 shows the actual model fragment that contains the bug (obtained from the oracle and considered the ground truth), while the right part of Fig. 12 shows the predicted model fragment output by the approach. The confusion matrix arranges the results of the comparison into four categories:

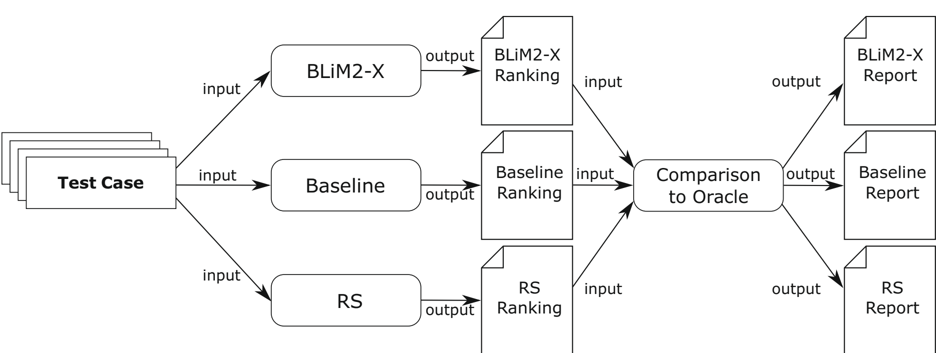


Fig. 11 Evaluation process

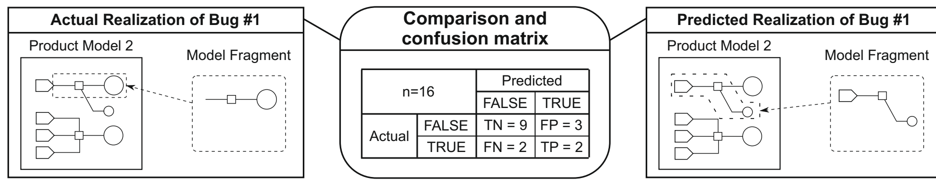


Fig. 12 Example of the comparison process and the confusion matrix

- True positive (TP): A model element present in the predicted model fragment that is also present in the actual model fragment (e.g., the upper power manager model element is a TP).
- True negative (TN): A model element not present in the predicted model fragment that is not present in the actual model fragment (e.g., the bottom inverter model element is a TN).
- False positive (FP): A model element present in the predicted model fragment that is not present in the actual model fragment (e.g., the upper inverter model element is a FP).
- False negative (FN): A model element not present in the predicted model fragment that is present in the actual model fragment (e.g., the upper inductor model element is a FN).

The confusion matrix holds the results of the comparison between the predicted results and the actual results. The result of the sum of all the categories (TP + TN + FP + FN) is the number of model elements (n) of the model that contains the predicted model fragment. However, in order to evaluate the performance of the approach, it is necessary to extract some measurements from the confusion matrix. 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 MCC) for each of the test cases for BLiM2-X and the baseline.

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

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \quad (1)$$

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

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \quad (2)$$

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

$$F\text{-measure} = 2 * \frac{\text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}} \quad (3)$$

Recall values can range between 0% (i.e., no single model element from the model fragment that contains the bug obtained from the oracle is present in any of the model frag-

ments of the solution) to 100% (i.e., all the model elements from the oracle are present in the solution).

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

However, none of these 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). MCC is a balanced measure which can be used even if the search space and the predicted solution are of very different sizes [9]. For this reason, MCC is one of the best measures for describing a confusion matrix [46]. It is defined as follows:

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

4.3 Case study

To evaluate the approach, we applied it to an industrial case study from our industrial partner.

4.3.1 BSH

The case study where we applied our evaluation process is the Induction Hob Product Family of our industrial partner (already presented in Sect. 2 as the running example). The oracle is composed of 46 induction hob models, which, on average, are composed of more than 500 elements. Our industrial partner provided us with documentation of 37 bug reports and the approved model fragments that contain the bugs.

The approved model fragments have between 3 and 15 model elements, with an average of 8 model elements. It is important to highlight that each model element has properties (that include terms), and a modification timespan, which are used to differentiate among model elements. Five domain engineers from our industrial partner were involved in providing the set of 37 bugs. The domain engineers of BSH based their selection on a combination of importance and frequency. The set of bugs provided are the most representatives of the bugs that occurs in BSH.

For each of the 37 bugs, we created a test case that includes the set of product models where that bug was manifested and a bug description, both obtained from the documentation. The following video illustrates the models and model fragments of BSH: www.youtube.com/watch?v=nS2sybEv6j0.

Each time we run each of the approaches, we get one results for a bug. As the approaches performs genetic operations, chance could affect the results. In order to minimize the effect of chance, we execute each of the approaches 30 times for each of the bugs as suggested in [6]. Then, for this case study, we executed 30 independent runs for each of the 37 test cases for BLiM2 and the baseline approach, i.e., 37 (bugs) × 10 (approaches) × 30 repetitions = 11,100 independent runs.

4.4 Results

In this section, we present the results obtained for the case study in BLiM2-X, the baseline, and RS approaches in BSH. Figure 13 shows the charts with the recall and precision results for the industrial domain. A dot in the graph represents the average result of precision and recall for each of the 37 bugs in BSH for the 30 repetitions.

Table 2 shows the mean values of recall, precision, the F-measure, and MCC for BLiM2-X, the baseline and the random search in the case study. There are five algorithms that obtained best results than the baseline. The BLiM2-3OT approach obtains the best results in recall, precision, and MCC, providing an average value of 89.84% in recall, 80.87% in precision, and 0.82 in MCC. The second best results are obtained by BLiM2-4O, providing an average value of 83.03% in recall, 75.05% in precision, and 0.76 in MCC. The third best values are obtained by BLiM2-2O, providing an average value of 80.76% in recall, 72.56% in precision, and 0.72 in MCC. The fourth best values are obtained by BLiM2-3OS, providing an average value of 73.72% in recall, 67.97% in precision, and 0.66 in MCC. The fifth best values are obtained by BLiM2-1O, providing an average value of 66.26% in recall, 61.29% in precision, and 0.59 in MCC. The rest of the algorithms obtained lower results. In terms of recall, precision, and MCC, BLiM2-3OT outperforms the rest of the approaches.

4.5 Statistical analysis

To properly compare our BLiM2 approaches and the baseline approach, all of the data resulting from the empirical analysis were analyzed using statistical methods following the guidelines in [5]. The goals of our statistical analysis are: (1) to provide formal and quantitative evidence (statistical significance) that BLiM2 does in fact have an impact on the comparison metrics (i.e., that the differences in the results were not obtained by mere chance); and (2) to show that those differences are significant in practice (effect size).

4.5.1 Statistical significance

To enable statistical analysis, all of the algorithms should be run a large enough number of times (in an independent

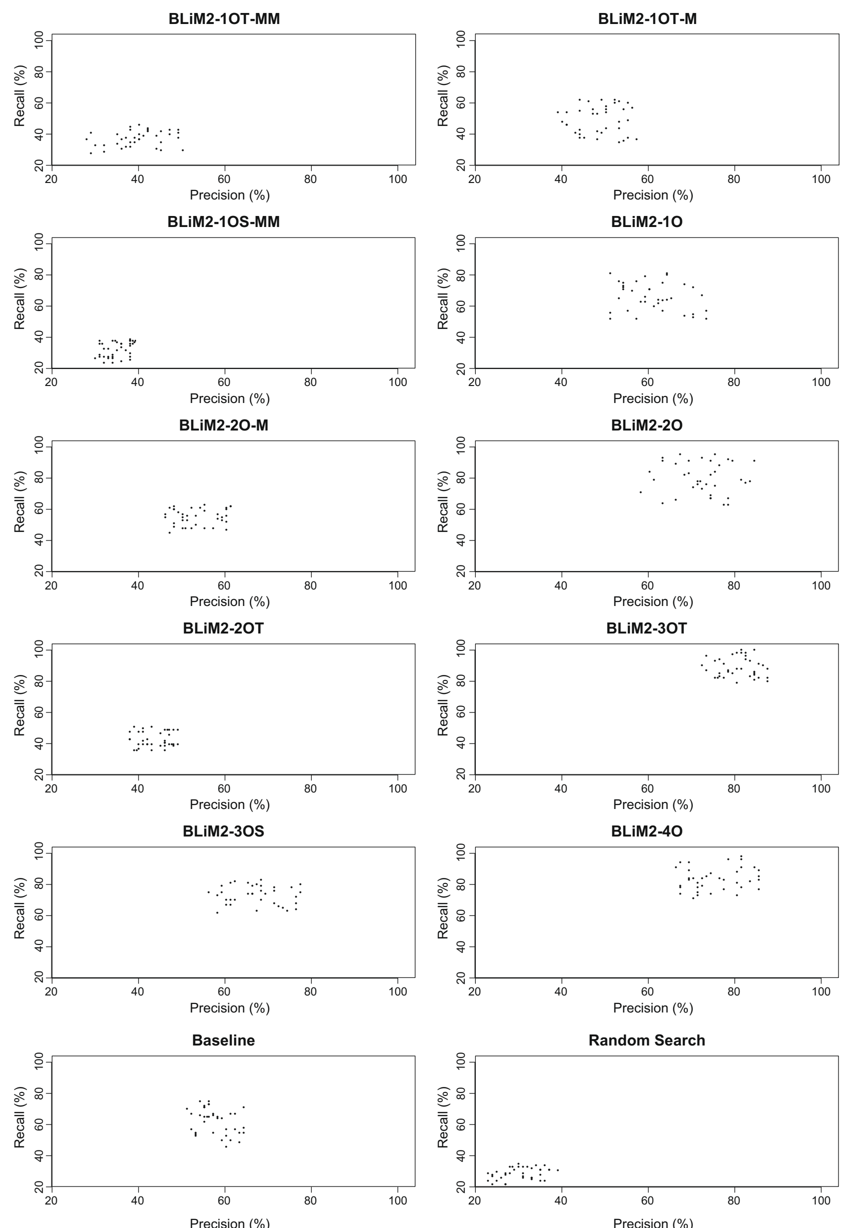


Fig. 13 Mean recall and precision values for BLiM2 and baseline approaches in BSH

Table 2 Mean values and standard deviations for recall, precision, the F-measure, and MCC in BSH

	Recall $\pm$ (σ)	Precision $\pm$ (σ)	F-measure $\pm$ (σ)	MCC $\pm$ (σ)
BLiM2-1OT-MM	37.98 ± 5.35	40.66 ± 6.61	37.20 ± 5.60	0.31 ± 0.05
BLiM2-1OT-M	49.11 ± 9.38	48.07 ± 5.40	48.53 ± 5.76	0.43 ± 0.06
BLiM2-1OS-MM	32.03 ± 5.57	35.91 ± 3.91	33.99 ± 3.19	0.25 ± 0.04
BLiM2-1O	66.26 ± 9.70	61.29 ± 7.11	63.43 ± 5.03	0.59 ± 0.05
BLiM2-2O-M	55.17 ± 5.41	53.02 ± 5.33	54.05 ± 4.69	0.48 ± 0.04
BLiM2-2O	80.76 ± 10.95	72.53 ± 7.68	75.34 ± 6.75	0.72 ± 0.07
BLiM2-2OT	43.06 ± 5.64	44.62 ± 3.24	43.82 ± 3.91	0.36 ± 0.04
BLiM2-3OT	89.84 ± 6.01	80.87 ± 4.51	84.18 ± 4.95	0.82 ± 0.04
BLiM2-3OS	73.72 ± 6.93	67.97 ± 6.91	70.39 ± 4.89	0.66 ± 0.05
BLiM2-4O	83.03 ± 7.43	75.05 ± 6.16	79.58 ± 5.66	0.76 ± 0.06
Baseline	62.10 ± 8.42	57.57 ± 4.89	59.49 ± 4.31	0.54 ± 0.04
Random Search	29.05 ± 4.90	30.20 ± 4.15	29.75 ± 3.77	0.21 ± 0.04

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

The statistical tests provide a probability value, p value. The p value 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 [6], so the hypothesis H_0 can be considered false.

The test that we must follow depends on the properties of the data. Since our data do 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 shows that it is more powerful than the others when working with real data [20]. In addition, according to Conover [10], the Quade test has shown better results than the others when the number of algorithms is low (no more than 4 or 5 algorithms).

The p values obtained in the test is $\ll 2.2 \times 10^{-16}$ for recall, precision, and MCC; the statistics values obtained are 82.581, 82.412, and 93.53 for recall, precision, and MCC, respectively. Since the p values are smaller than 0.05 for recall, precision, and MCC, we reject the null hypothesis. Consequently, we can state that there are differences among the algorithms for the performance indicators of recall, precision, and MCC.

However, with the Quade test, we cannot answer the following question: Which of the algorithms gives the best performance? In this case, the performance of each algorithm should be individually compared against all other alternatives. In order to do this, we perform an additional post hoc analysis. This kind of analysis performs a pair-wise comparison among the results of each algorithm, determining whether statistically significant differences exist among the results of a specific pair of algorithms.

Table 3 shows the p values of Holm's post hoc analysis for the case study and the performance indicators for the algorithm that obtains the best results, BLiM2-3OT. [1] The majority of the p values obtained are smaller than their corresponding significance threshold value (0.05), indicating that the differences in performance between the algorithms are significant. However, when we compare BLiM2-3OT with BLiM2-2O (sixth row), and BLiM2-3OT with BLiM2-4O, the values are greater than the threshold. This indicates that the differences between those algorithms could be due to the stochastic nature of the algorithms and are not significant.

4.5.2 Effect size

When comparing algorithms with a large enough number of runs, statistically significant differences can be obtained even if they are so small as to be of no practical value [6]. Thus, it is important to assess whether an algorithm is statistically better than another and to assess the magnitude of the improvement. Effect size measures are needed to analyze this.

For a nonparametric effect size measure, we use Vargha and Delaney's $\hat{A}_{12}$ [22,58]. $\hat{A}_{12}$ measures the probability that running one algorithm yields higher values than run-

[1] Although we have performed the entire statistical significance analysis, here we decide to show only the combinations with the algorithm that obtained the best results. Table 5 shows the entire table with the sixty-six combinations (at the end of the paper).

Table 3 Holm's post hoc p values and the $\hat{A}_{12}$ statistic for each pair of algorithms

	Holm's			$\hat{A}_{12}$		
	Recall	Precision	MCC	Recall	Precision	MCC
3OT vs. 1OT-MM	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	1	1	1
3OT vs. 1OT-M	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	1	1	1
3OT vs. 1OS-MM	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	1	1	1
3OT vs. 1O	$5.8x10^{-10}$	$1.6x10^{-8}$	$1.4x10^{-11}$	0.99489	0.99672	1
3OT vs. 2O-M	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	1	1	1
3OT vs. 2O	0.21685	0.00743	0.00940	0.75347	0.85793	0.89883
3OT vs. 2OT	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	1	1	1
3OT vs. 3OS	$7.1x10^{-5}$	0.00066	$7.0x10^{-5}$	0.98247	0.95435	0.99927
3OT vs. 4O	0.68994	0.57992	0.45910	0.70380	0.73887	0.79657
3OT vs. Baseline	$4.8x10^{-12}$	$1.6x10^{-12}$	$\ll 2.2x10^{-16}$	1	1	1
3OT vs. RS	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	1	1	1

ning another algorithm. If the two algorithms are equivalent, then $\hat{A}_{12}$ will be 0.5.

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

Table 3 shows the values of the size effect statistics.[2] In general, the largest differences were obtained between BLiM2-3OT and BLiM2-1OT-MM, BLiM2-1OT-M, BLiM2-1OS-MM, BLiM2-2O-M, BLiM2-2OT, the baseline, and the RS, where BLiM2-3OT achieves better results all of the times. When comparing BLiM2-3OT and BLiM2-4O, the differences are not so large, with around 78% in performance.

BLiM2-3OT obtained the best performance results among the twelve evaluated approaches (see Table 2). The performed statistical analysis indicated that BLiM2-3OT outperforms the rest of the approaches in terms of recall, precision, and MCC. Overall, these results confirm that the use of BLiM2-3OT against the baseline approach has an actual impact. In other words, BLiM2-3OT obtained better results in recall, which means that the model fragment proposed as the solution has more relevant elements for the bug that must be located than the model fragments proposed by the baseline approach. In the same way, BLiM2-3OT obtained better results in precision, which means that the model fragment proposed as the solution has less non-relevant elements for the bug that we must be located than the model fragments proposed by the baseline approach.

4.6 Discussion

Our results confirm that the BLiM2 variants and the baseline approaches are better than random search based on the four metrics (recall, precision, F-measure, and MCC) on the BSH case study. Through this study, we concluded that there is empirical evidence to support the significance of the results of our algorithms. Thus, an intelligent algorithm is required to find good solutions to perform bug localization in models.

The BLiM2-3OT variant outperforms the rest of the variants of our approach. However, it did not obtain the perfect solution for a bug in any of the cases. In other words, the model fragments obtained from our approach do not include all of the model elements that contain the bug.

One of the issues that we detected that cause this outcome is the vocabulary mismatch. This means that for a specific concept, the terms used in the bug description are different from the terms used in the models. For example, the bug description includes the word *'current'* to refer to the electrical energy that reaches an inductor. However, in the models, the word *'power'* stands for the same concept. Nevertheless, this issue could be solved by augmenting the natural language processing (NLP) with a dictionary of synonyms. In the same way, we have also detected cases in which in-house terms are used; for example, instead of using the word *inverter*, the name of a manufacturer is used (*'Fairchild'*). Therefore, the regular dictionary of synonyms would not work in this case. This suggests that the dictionary of synonyms should be refined by domain engineers to include in-home terms.

Another issue is the case in which the bug description is incomplete. For example, in a bug description the following sentence can appear: *'The induction hob crashes when*

[2] Although we have performed the entire size effect statistics analysis, here we decide to show only the combinations with the algorithm that obtained the best results. Table 6 shows the entire table with the sixty-six combinations (at the end of the paper).

the user selects the power level 9 for a double inductor'. The engineers understand that the inductors have to reach that power level when the user selects it. However, this sentence also embodies implicit knowledge that is not written but is obvious to the domain engineers: *'The double inductor is formed by two concentric inductors'*, and *'If the pot that is on the top is large enough, both inductors have to reach the power level 9, otherwise, only the central inductor must reach that level.'*. Omitting words in the bug description negatively influences the fitness value of textual similarity since the fitness value of textual similarity is based on the co-occurrence of terms. This suggests that we must make the engineers aware of this issue. They should know that in cases in which the results obtained do not have enough quality and more model elements need to be located, they can reformulate the descriptions of the bugs making the implicit knowledge explicit.

In addition, the variants of our approach that combine model and metamodel terms in one objective obtain better results. This outcome is due to the fact that, in the bug description, terms from the model and the metamodel are mixed. From the metamodel comes general terms that are shared by all the induction hobs, but differentiate the parts. For example, with the words from the metamodel, the approach discriminates between inductors and inverters. From the model come terms that specifies the part of the induction hob. For example, with the words from the model, the approach discriminates between double inductor and single inductor. A complete example is the following sentence from a bug description: *'In the induction hobs that have a triple inductor and a pool inductor, the power booster does not work'*. The terms from the metamodel are: *induction hob, inductor, and power*, and the terms from the model are: *triple, pool, and booster*. Therefore, higher values of textual similarity (co-occurrence of terms) will be reached when comparing the description of the bug with the combined terms of the model and the metamodel.

Therefore, the variants of our approach that combine model and metamodel terms in one objective obtains similar results. However, the differences come from the way in which the Defect Localization Principle is measure. For example, between BLiM2-3OT and BLiM2-2O the difference is around 10% in favor of BLiM2-3OT. In BLiM2-2O, the approach gives one fitness value for the timespan, this produces that there are not so many differences between those that have recently been modified and those that have not.

We believe that the above is especially interesting since it can be an important difference between generic modeling languages and DSLs when locating bugs. Unlike generic modeling languages, such as UML, metamodel elements in DSLs contain domain information. On the one hand, the metaelement *Class* of UML is generic enough to be relevant for different domains. On the other hand, a metaelement such as *Inductor* of the induction hobs DSL is relevant for specific domains only. This is also the case of other generic modeling languages, such as BPMN or ARCHIMATE. This suggests that in the hypothetical case that the induction hobs would have been specified with a generic modeling language, the results could have been worse since the terms of the model (e.g., class) would not be similar to the terms of the description of the bug (e.g., inductor). Therefore, we think that specific experiments should be carried out with generic modeling languages to determine the performance with the current approach, assess the need to reformulate the query, or even identify new objectives to guide the bug localization.

Finally, our results confirm the relevance of the Defect Localization Principle to models. The majority of bugs provided by the industrial partner (about 90%) are related to recent modifications. For bugs that are not related to recent modifications, our approach (in spite of including a timespan objective) obtains similar or slightly worse results than the baseline in terms of recall, precision, and MCC. This suggests that, given a bug description where we do not know whether or not the recent modification timespans are relevant, the inclusion of the objective of the Defect Localization Principle, in general, leads to better results than if it is not included. Since modification timespan is important in the localization of bugs, the variants of our approach in which we give more relevance to it obtain better results.

The ten variants of our approach are vulnerable to the following issues: vocabulary mismatch and descriptions with implicit knowledge. For textual similarity, the variants that mix the information from the model and the metamodel (BLiM2-2O and BLiM2-3OT) obtained the best results. For timespan modification, the variants that differentiate between modifications in the model and the metamodel (BLiM2-2OT and BLiM2-4O) obtained the best results. BLiM2-3OT, which combines the above, is the one that obtains the best results in models like those of our industrial partner.

Bugs cannot be located using only textual similarity (as the baseline does) due to the vocabulary mismatch and to the descriptions with implicit knowledge. The reason is that some text is missing or that the text is different between the description of the bug and the models. Bugs cannot be located using only the Default Localization Principle either. The reason is that all recent modifications would be suggested as relevant to the bug. Bugs can not be located using only information from the metamodel. The reason is the terms of the metamodel are the same for the specific elements of the model, and for the timespan occurs the same, if a modification is performed in an element of the metamodel the time is the same for all those elements of the models. The combination of model and metamodel information allows the approach to discriminate between the rest of the element and between the specific elements.

Hence, the combination of textual similarity and Defect Localization Principle pays off to locate bugs. Specifically, the combination that gives more weight to the Defect Localization Principle than to the textual similarity. In particular, the best results are obtained by the variant BLiM2-3OT of our approach that has 2 objectives of 3 related to the Defect Localization Principle and 1 of 3 related to textual similarity. It turns out that with 2 of 3 objectives the fragments are prioritized according to the Defect Localization Principle (it happens to about 90% of the bugs) and the third objective (even imperfect due to the vocabulary mismatch and to the descriptions with implicit knowledge) is able to differentiate between the recent modifications not relevant to the bugs and those that are relevant to the bugs.

4.7 Threats to validity

In this section, we present some of the threats to validity. We follow the guidelines suggested by De Oliveira et. al [11] to identify those that are applicable to this work.

Conclusion validity threats: The first identified threat of this type is *not accounting for random variation*. To address this threat, we considered 30 independent runs for each bug with each algorithm. The second threat is *lack of formal hypothesis and statistical tests*. In this paper, we employed standard statistical analysis following accepted guidelines [6] in order to avoid this threat. The third threat is the *lack of good descriptive analysis*. In this work, we have used precision, recall, the F-measure, and MCC metrics to analyze the confusion matrix obtained from the experiments; however, other metrics could also be applied. In addition, some works argue that the use of the Vargha and Delaney's $\hat{A}_{12}$ metric may be misrepresentative [43] and that the data should be pretransformed before applying it. We did not find any use case for data pretransformation that applies to our case studies.

Internal validity threats: The first identified threat of this type is *poor parameter settings*. In this paper, we used standard values for the algorithms. These values have been tested in similar algorithms for feature localization [36]. In addition, the choice of the k value in the application of SVD can produce suboptimal accuracy when using LSI for software artifacts [44]. However, we plan to evaluate all of the parameters of our algorithm in a future work. The second threat is *lack of clear of data collection tools and procedures*. The set of 37 bugs used in the evaluation has been provided by domain experts of BSH. The set of bugs provided are the most representatives of the bugs that occurs in BSH. The third threat is *lack of real problem instances*. The evaluation of this paper was applied to an industrial case study, BSH.

Construct validity threats: The identified threat is *lack of assessing the validity of cost measures*. To address this threat, we performed a fair comparison between BLiM2-X and the baseline by generating the same number of model fragments and using the same number of fitness evaluations.

External validity threats: The identified threat of this type is *lack of a clear object selection strategy*. This threat is addressed by using an industrial case study. Our instances are collected from real-world problems.

5 Related work

In recent years, many bug localization approaches have been proposed. These approaches are usually IR-based approaches, and some of them add the Defect Localization Principle. Since our bug localization in models approach applies these techniques to models, in this section, we review some relevant works in the literature. Table 4 shows a summary of the main features of these works.

The first block of works in Table 4 shows the works that are related to this paper:

- Kusumoto et al. [29] and Liang et al. [34] apply static program slicing to bug localization. They apply static slicing to reduce the search domain while programmers locate bugs in their programs. In [29], they evaluate this technique and confirm that it is useful for bug localization. In [34], they use this technique to improve the efficiency of data flow analysis in the presence of pointer variables.
- Mao et al. [39] use dynamic slicing and statistical bug localization. They utilize program slices of a set of test runs to capture the influence of a program entity's execution on the output, and they use statistical analysis to measure the level of suspiciousness of each program entity being faulty. Their approach is called approximate dynamic backward slice.
- In the same way, Alves et al. [2] combine dynamic slicing and spectrum-based techniques. They rank all of the statements in a program based on their level of suspiciousness, which is calculated by using a spectrum-based technique (the Tarantula technique). Then, they generate a dynamic slice with respect to a failure-indicating variable at the failure point. The statements that are not in the slice are removed from the ranking to reduce the search domain.
- Gong et al. [21] propose an interactive localization technique called TALK. This approach incorporates programmers' feedback into spectrum-based fault localization techniques. When the programmer receives the ranking of program elements that can cause the bug, he or she can judge the correctness of each element and provide this information as feedback to reorder the ranking.
- Saha et al. [50], Zhou et al. [65], and Rao et al. [48] apply information retrieval for bug localization. In [50], the authors present BLUiR, Bug Localization

Table 4 Summary of the works cited in the Related Work Section

Approach	Information retrieval	Defect Localization Principle	Locates	Source code	Model	Metamodel
Kusumoto et al. [29]	No	No	Bugs	Yes	No	No
Liang et al. [34]	No	No	Bugs	Yes	No	No
Mao et al. [39]	No	No	Bugs	Yes	No	No
Alves et al. [2]	No	No	Bugs	Yes	No	No
Gong et al. [21]	No	No	Bugs	Yes	No	No
Saha et al [50]	Yes	No	Bugs	Yes	No	No
Zhou et al. [65]	Yes	No	Bugs	Yes	No	No
Rao et al. [48]	Yes	No	Bugs	Yes	No	No
Lukins et al. [38]	Yes	No	Bugs	Yes	No	No
Kim et al. [28]	Yes	No	Bugs	Yes	No	No
Le et al. [32]	Yes	No	Bugs	Yes	No	No
Hoang et al. [25]	Yes	No	Bugs	Yes	No	No
Lam et al. [30]	Yes	No	Bugs	Yes	No	No
Zamani et al. [62]	Yes	Yes	Bugs	Yes	No	No
Sisman and Kak [54]	Yes	Yes	Bugs	Yes	No	No
Wang and Lo [59]	Yes	Yes	Bugs	Yes	No	No
Wille et al. [60]	No	No	Features	No	Yes	No
Holthusen et al. [26]	No	No	Features	No	Yes	No
Zhang et al. [63,64]	No	No	Features	No	Yes	No
Martinez et al. [41,42]	No	No	Features	No	Yes	No
Lopez-Herrejon et al. [37]	No	No	Features	No	Yes	No
Arcega et al. [3]	Yes	No	Features	No	Yes	No
Font et al. [18,19]	Yes	No	Features	No	Yes	No
Arcega et al. [4]	Yes	Yes	Bugs	No	Yes	No
Our approach	Yes	Yes	Bugs	No	Yes	Yes

Using information Retrieval. In [65], the authors propose BugLocator. Both works use an initial bug report to rank the source code files in descending order based on their relevance to the bug report.
– Lukins et al. [38] used latent Dirichlet allocation (LDA) for predicting the location of a newly reported bug. They use source code comments and identifiers as information resources for predicting the locations of bugs.
– Kim et al. [28] propose both a one-phase and a two-phase prediction model to recommend files to fix. In the one-phase model, they create features from textual information and metadata of bug reports, apply Nave Bayes to train the model using previously fixed files as classification labels, and then use the trained model to assign multiple source files to a bug report. In the two-phase model, they first apply their one-phase model to classify a new bug report as either 'predictable' or 'deficient' and then make predictions only for 'predictable' report.
– Le et al. [32] and Hoang et al. [25] combine information retrieval and spectrum-based techniques. In [25,32], they present two approaches that utilize multimodal information from both bug reports and program spectra to localize bugs.
– Lam et al. [30] present an approach that uses deep neural network (DNN) in combination with rVSM, an information retrieval technique. rVSM collects the feature on the textual similarity between bug reports and source files. DNN is used to learn related terms in bug reports to potentially different code tokens and terms in source files.
– Zamani et al. [62], Sisman and Kak [54], and Wang and Lo [59] include the Defect Localization Principle in their approaches. In [62], they proposed an approach that included weighting and ranking the source code locations based on both the textual similarity with a change request and the use of the time metadata. In [54], they utilize time decay in weighting the files in a probabilistic information retrieval model. In [59], they present AmaLgam+, which is a method for locating relevant buggy files that puts together fives sources of information: version history, similar reports, structure, stack traces, and reporter information.
These approaches give better results than information retrieval techniques.

All of the above works present approaches that only take into account the source code as the artifact that represents the bug. These approaches have not been applied to models. In contrast, we propose an approach that can be configured to apply these ideas (static and dynamic analysis, information retrieval, and the Defect Localization Principle) in models. We have evaluated the approach successfully by applying them in models. Moreover, our approach does not apply the dynamic analysis that is used by some of those cited. Our future work involves designing an approach that addresses the dynamic analysis idea using models at runtime [7]

Some works focus on the localization 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:

- Wille et al. [60] present an approach where the similarity between models is measured following an exchangeable metric, taking into account different attributes of the models. Then, the approach is further refined [26] to reduce the number of comparisons needed to mine the family model.
- The authors in [63] propose a generic approach to automatically compare products and locate the feature realizations in terms of a CVL model. In [64], the approach is refined to automatically formalize the feature realizations of new product models that are added to the system.
- Martinez et al. [42] propose an extensible approach that is based on comparisons to extract the feature formalization on a family of models. In addition, they provide means to extend the approach to locate features in any kind of asset based on comparisons.
- The MoVaPL approach [41] considers the identification of variability and commonality in model variants as well as the extraction of a Model-based Software Product Line (MSPL) from the features identified in these s. MoVaPL builds on a generic representation of models making it suitable for any MOF-based models.
- Lopez-Herrejon et al. [37] evaluate three standard search-based techniques (evolutionary algorithm, hill climbing, and random search) in order to calculate the relationships of a feature model. They calculate the feature relationships of the feature specification layer, while our work locates the model fragments of the product realization

Nevertheless, all of these approaches are based on mechanical comparisons among the models, classifying the elements based on their similarity, and identifying the dissimilar elements as the feature realizations. In contrast, our work does not rely on model comparisons to locate the bugs. Specifically, in our work, we use a multi-objective evolutionary algorithm that uses both the similarity to the bug report and the timespan weighting as fitness functions.

In addition, there are other approaches for feature localization in models that, although they were not designed to locate bugs, could potentially be applied to do this. In the second block of works in Table 4, we show our previous works.

- In [3], we present an approach that is composed of dynamic analysis and information retrieval at the model level. We compare different ways of creating model traces. This work outperforms the feature localization in source code.
- Font et al. [18,19] propose two approaches that use evolutionary algorithms to locate features in a model. This work does not take into account the Defect Localization Principle for bug localization.
- In [4], we analyze the influence of several timespan weightings on bug localization in models. We evaluate four timespan weightings: the most recent model modifications, the oldest model modifications, the mean of the modification timespan of the modified model elements, and the sum of the modification timespan of the modified model elements. The results show that the use of the most recent timespan model modifications provides the best results in bug localization.

In this work, we adapt the Defect Localization Principle that is used in source code to models. The approach of this paper supports ten different fitness functions, in which the Defect Localization Principle is used. This principle is applied using the most recent timespan model modifications because it obtains the best results in [4]. In addition, in contrast to our previous works, our approach takes into account the domain information from the metamodel.

6 Conclusion

Bug localization is a significant maintenance activity. In this paper, we have proposed an approach for bug localization in models (BLiM2) which enables us to evaluate to what extent the ideas that have been successfully applied in source code for bug localization (textual similarity to bug description and Defect Localization Principle) also work in models. In addition, BLiM2 takes information from two levels, the model and the metamodel, taking advantage of a particularity of the models that does not exist in source code (in the models, there is domain information in both the model and the metamodel).

We evaluated our BLiM2 approach in an industrial case study, BSH, and compared it with the approach that they are using for bug localization. We determined which approach produces the best results in terms of precision, recall, the F-measure, and MCC. To do this, we applied the approaches

Table 5 Holm's post hoc p values for recall for each pair of algorithms

	1OT-MM	1OT-M	1OS-MM	1O	2O-M	2O	2OT	3OT	3OS	4O	Baseline
Recall											
1OT-M	0.01218	–	–	–	–	–	–	–	–	–	–
1OS-MM	0.22381	$8.4x10^{-7}$	–	–	–	–	–	–	–	–	–
1O	1.2×10^{-14}	1.1×10^{-5}	$\ll 2.2 \times 10^{-16}$	–	–	–	–	–	–	–	–
2O-M	2.1×10^{-6}	0.27833	3.5×10^{-12}	0.03263	–	–	–	–	–	–	–
2O	$\ll 2.2 \times 10^{-16}$	$\ll 2.2 \times 10^{-16}$	$\ll 2.2 \times 10^{-16}$	0.00013	5.9×10^{-12}	–	–	–	–	–	–
2OT	0.65083	0.35431	0.00340	3.3×10^{-10}	0.00171	$\ll 2.2 \times 10^{-16}$	–	–	–	–	–
3OT	$\ll 2.2 \times 10^{-16}$	$\ll 2.2 \times 10^{-16}$	$\ll 2.2 \times 10^{-16}$	5.8×10^{-10}	$\ll 2.2 \times 10^{-16}$	0.21685	$\ll 2.2 \times 10^{-16}$	–	–	–	–
3OS	$\ll 2.2 \times 10^{-16}$	4.6×10^{-11}	$\ll 2.2 \times 10^{-16}$	0.22577	5.7×10^{-6}	0.21685	$\ll 2.2 \times 10^{-16}$	7.1×10^{-5}	–	–	–
4O	$\ll 2.2 \times 10^{-16}$	$\ll 2.2 \times 10^{-16}$	$\ll 2.2 \times 10^{-16}$	1.4×10^{-6}	6.5×10^{-15}	0.93749	$\ll 2.2 \times 10^{-16}$	0.68994	0.01387	–	–
Baseline	2.4×10^{-12}	0.00031	$\ll 2.2 \times 10^{-16}$	0.93749	0.21685	4.4×10^{-6}	2.8×10^{-8}	4.8×10^{-12}	0.04335	2.3×10^{-8}	–
RS	0.01795	2.7×10^{-9}	0.93749	$\ll 2.2 \times 10^{-16}$	2.9×10^{-15}	$\ll 2.2 \times 10^{-16}$	5.7×10^{-5}	$\ll 2.2 \times 10^{-16}$	$\ll 2.2 \times 10^{-16}$	$\ll 2.2 \times 10^{-16}$	$\ll 2.2 \times 10^{-16}$
Precision											
1OT-M	0.00650	–	–	–	–	–	–	–	–	–	–
1OS-MM	0.43504	2.4×10^{-6}	–	–	–	–	–	–	–	–	–
1O	$\ll 2.2 \times 10^{-16}$	4.6×10^{-7}	$\ll 2.2 \times 10^{-16}$	–	–	–	–	–	–	–	–
2O-M	4.7×10^{-7}	0.32239	6.3×10^{-12}	0.00650	–	–	–	–	–	–	–
2O	$\ll 2.2 \times 10^{-16}$	6.0×10^{-15}	$\ll 2.2 \times 10^{-16}$	0.05708	1.1×10^{-8}	–	–	–	–	–	–
2OT	0.43504	0.43504	0.00625	7.2×10^{-12}	0.00137	$\ll 2.2 \times 10^{-16}$	–	–	–	–	–
3OT	$\ll 2.2 \times 10^{-16}$	$\ll 2.2 \times 10^{-16}$	$\ll 2.2 \times 10^{-16}$	1.6×10^{-8}	$\ll 2.2 \times 10^{-16}$	0.00743	$\ll 2.2 \times 10^{-16}$	–	–	–	–
3OS	$\ll 2.2 \times 10^{-16}$	8.4×10^{-13}	$\ll 2.2 \times 10^{-16}$	0.32239	4.8×10^{-7}	0.57992	$\ll 2.2 \times 10^{-16}$	0.00066	–	–	–
4O	$\ll 2.2 \times 10^{-16}$	$\ll 2.2 \times 10^{-16}$	$\ll 2.2 \times 10^{-16}$	2.5×10^{-5}	1.1×10^{-14}	0.32239	$\ll 2.2 \times 10^{-16}$	0.57992	0.06481	–	–
Baseline	1.2×10^{-12}	0.00045	$\ll 2.2 \times 10^{-16}$	0.57992	0.32239	0.00044	5.8×10^{-8}	1.6×10^{-12}	0.00616	1.2×10^{-8}	–
RS	0.01619	9.8×10^{-10}	0.57992	$\ll 2.2 \times 10^{-16}$	3.1×10^{-16}	$\ll 2.2 \times 10^{-16}$	2.0×10^{-5}	$\ll 2.2 \times 10^{-16}$	$\ll 2.2 \times 10^{-16}$	$\ll 2.2 \times 10^{-16}$	$\ll 2.2 \times 10^{-16}$
MCC											
1OT-M	0.0076	–	–	–	–	–	–	–	–	–	–
1OS-MM	0.1407	1.6×10^{-7}	–	–	–	–	–	–	–	–	–
1O	$\ll 2.2 \times 10^{-16}$	9.1×10^{-7}	$\ll 2.2 \times 10^{-16}$	–	–	–	–	–	–	–	–
2O-M	2.6×10^{-6}	0.4540	1.2×10^{-12}	0.0042	–	–	–	–	–	–	–
2O	$\ll 2.2 \times 10^{-16}$	$\ll 2.2 \times 10^{-16}$	$\ll 2.2 \times 10^{-16}$	0.0010	2.2×10^{-12}	–	–	–	–	–	–
2OT	0.4591	0.4540	0.0015	8.4×10^{-12}	0.0021	$\ll 2.2 \times 10^{-16}$	–	–	–	–	–
3OT	$\ll 2.2 \times 10^{-16}$	$\ll 2.2 \times 10^{-16}$	$\ll 2.2 \times 10^{-16}$	1.4×10^{-11}	$\ll 2.2 \times 10^{-16}$	0.0094	$\ll 2.2 \times 10^{-16}$	–	–	–	–
3OS	$\ll 2.2 \times 10^{-16}$	3.2×10^{-14}	$\ll 2.2 \times 10^{-16}$	0.0637	8.7×10^{-9}	0.4591	$\ll 2.2 \times 10^{-16}$	7.0×10^{-5}	–	–	–
4O	$\ll 2.2 \times 10^{-16}$	$\ll 2.2 \times 10^{-16}$	$\ll 2.2 \times 10^{-16}$	3.5×10^{-7}	$\ll 2.2 \times 10^{-16}$	0.4591	$\ll 2.2 \times 10^{-16}$	0.4591	0.0309	–	–
Baseline	1.1×10^{-10}	0.0042	$\ll 2.2 \times 10^{-16}$	0.4540	0.4591	1.2×10^{-7}	7.5×10^{-7}	$\ll 2.2 \times 10^{-16}$	7.8×10^{-5}	3.2×10^{-12}	–
RS	0.0047	1.6×10^{-10}	0.4591	$\ll 2.2 \times 10^{-16}$	2.6×10^{-16}	$\ll 2.2 \times 10^{-16}$	9.0×10^{-6}	$\ll 2.2 \times 10^{-16}$	$\ll 2.2 \times 10^{-16}$	$\ll 2.2 \times 10^{-16}$	$\ll 2.2 \times 10^{-16}$

Table 6 $\hat{A}_{12}$ statistic for each pair of algorithms

↓ vs. →	1OT-MM	1OT-M	1OS-MM	1O	2O-M	2O	2OT	3OT	3OS	4O	Baseline
Recall											
1OT-M	0.85866	–	–	–	–	–	–	–	–	–	–
1OS-MM	0.24690	0.04456	–	–	–	–	–	–	–	–	–
1O	1	0.89664	1	–	–	–	–	–	–	–	–
2O-M	0.99891	0.68627	1	0.16143	–	–	–	–	–	–	–
2O	1	1	1	0.83638	0.99927	–	–	–	–	–	–
2OT	0.76479	0.29401	0.95435	0	0.05917	0	–	–	–	–	–
3OT	1	1	1	0.99489	1	0.75347	1	–	–	–	–
3OS	1	0.99890	1	0.72498	0.99744	0.30241	1	0.01753	–	–	–
4O	1	1	1	0.93755	1	0.60189	1	0.29620	0.85757	–	–
Baseline	0.99963	0.84697	1	0.39262	0.75566	0.09533	0.98319	0	0.15194	0.01278	-
RS	0.09204	0.00037	0.27904	0	0	0	0	0	0	0	0
Precision											
1OT-M	0.8656	–	–	–	–	–	–	–	–	–	–
1OS-MM	0.2297	0.00037	–	–	–	–	–	–	–	–	–
1O	1	0.93644	1	–	–	–	–	–	–	–	–
2O-M	0.9661	0.73411	1	0.19065	–	–	–	–	–	–	–
2O	1	1	1	0.88093	0.96676	–	–	–	–	–	–
2OT	0.7141	0.23192	0.98722	0	0.04602	0	–	–	–	–	–
3OT	1	1	1	0.99671	1	0.85793	1	–	–	–	–
3OS	1	0.99890	1	0.74981	0.95617	0.30131	1	0.04565	–	–	–
4O	1	1	1	0.93353	1	0.60738	1	0.26114	0.80022	–	–
Baseline	1	0.91965	1	0.35245	0.72899	0.03324	1	0	0.09788	0	–
RS	0.10811	0.00037	0.21110	0	0	0	0.00292	0	0	0	0
MCC											
1OT-M	0.91563	–	–	–	–	–	–	–	–	–	–
1OS-MM	0.20088	0.00292	–	–	–	–	–	–	–	–	–
1O	1	0.97918	1	–	–	–	–	–	–	–	–
2O-M	0.99598	0.77283	1	0.05696	–	–	–	–	–	–	–
2O	1	1	1	0.94668	1	–	–	–	–	–	–
2OT	0.77356	0.19722	0.98685	0	0.01534	0	–	–	–	–	–
3OT	1	1	1	1	1	0.89883	1	–	–	–	–
3OS	1	1	1	0.85646	0.99708	0.24507	1	0.00073	–	–	–
4O	1	1	1	0.99270	1	0.66983	1	0.20343	0.91271	–	–
Baseline	1	0.94595	1	0.24653	0.84222	0.01169	1	0	0.03287	0	–
RS	0.07779	0	0.24836	0	0	0	0	0	0	0	0

in BSH that has a model-based product family (firmware of induction hobs). We present our evaluation, which includes the following: experimental setup, results, statistical analysis, and threats to validity.

The results show that the domain information from the metamodel pays off for bug localization. Specifically, the BLiM2-3OT of our approach achieves the best results. It has three objectives in the fitness function: one that combines information from the model and the metamodel for text similarity; one that takes into account the modification timespan of the model; and one that takes into account the modification timespan of the metamodel. Results also show that our approach can be applied in real-world environments. The statistical analysis of the results provides evidence of their significance.

Acknowledgements This work has been partially supported by the Ministry of Economy and Competitiveness (MINECO) through the Spanish National R+D+i Plan and ERDF funds under the project Model-Driven Variability Extraction for Software Product Line Adoption (TIN2015-64397-R). We also thank ITEA3 15010 REVaMP2 Project.

References

1. Apache opennlp: Toolkit for the processing of natural language text. http://opennlp.apache.org/ (2010). Online; Accessed 04 April 2017
2. Alves, E., Gligoric, M., Jagannath, V., d'Amorim, M.: Fault-localization using dynamic slicing and change impact analysis. In: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE '11, pp. 520–523. IEEE Computer Society, Washington, DC, USA (2011). https://doi.org/10.1109/ASE.2011.6100114
3. Arcega, L., Font, J., Haugen, Ø., Cetina, C.: On the influence of models at run-time traces in dynamic feature location. In: Modelling Foundations and Applications - 13th European Conference, ECMFA 2017, Held as Part of STAF 2017, Marburg, Germany, July 19–20, 2017, Proceedings (2017)
4. Arcega, L., Font, J., Haugen, Ø., Cetina, C.: On the influence of modification timespan weightings in the location of bugs in models. In: Proceedings of the 26th International Conference on Information Systems Development, ISD 2017, Larnaca, Cyprus, September 6–8, 2017 (2017)
5. Arcuri, A., Briand, L.: A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. Softw. Test. Verif. Reliab. **24**(3), 219–250 (2014). https://doi.org/10.1002/stvr.1486
6. Arcuri, A., Fraser, G.: Parameter tuning or default values? An empirical investigation in search-based software engineering. Empir. Softw. Eng. **18**(3), 594–623 (2013). https://doi.org/10.1007/s10664-013-9249-9
7. Bencomo, N., Hallsteinsen, S., de Almeida, E.Santana: A view of the dynamic software product line landscape. Computer **45**(10), 36–41 (2012). https://doi.org/10.1109/MC.2012.292
8. Blei, D.M., Ng, A.Y., Jordan, M.I., Lafferty, J.: Latent dirichlet allocation. J. Mach. Learn. Res. **3**(4/5), 993–1022 (2003)
9. Boughorbel, S., Jarray, F., El-Anbari, M.: Optimal classifier for imbalanced data using Matthews correlation coefficient metric. PLOS ONE **12**(6), 1–17 (2017). https://doi.org/10.1371/journal.pone.0177678
10. Conover, W.J.: Practical Nonparametric Statistics, 3rd edn. Wiley, Hoboken (1999)
11. de Oliveira Barros, M., Dias-Neto, A.C.: 0006/2011-threats to validity in search-based software engineering empirical studies. RelaTe-DIA **5**(1), (2011)
12. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: Nsga-ii. Trans. Evol. Comput. **6**(2), 182–197 (2002). https://doi.org/10.1109/4235.996017
13. Deerwester, S., Dumais, S.T., Furnas, G.W., Landauer, T.K., Harshman, R.: Indexing by latent semantic analysis. J. Am. Soc. Inf. Sci. **41**(6), 391–407 (1990). https://doi.org/10.1002/(SICI)1097-4571(199009)41:6<391::AID-ASI1>3.0.CO;2-9
14. Dit, B., Revelle, M., Gethers, M., Poshyvanyk, D.: Feature location in source code: a taxonomy and survey. J. Softw. Maint. Evol. Res. Pract. (2011)
15. Dyer, D.W.: The watchmaker framework for evolutionary computation (evolutionary/genetic algorithms for java). http://watchmaker.uncommons.org/ (2006). Online; Accessed 04 April 2017
16. Efficient Java Matrix Library. https://ejml.org (2016). Online; Accessed 04 April 2017
17. Font, J., Arcega, L., Haugen, O., Cetina, C.: Leveraging variability modeling to address metamodel revisions in model-based software product lines. Comput. Lang. Syst. Struct. **48**, 20–38 (2017). https://doi.org/10.1016/j.cl.2016.08.003
18. Font, J., Arcega, L., Haugen, Ø., Cetina, C.: Feature location in model-based software product lines through a genetic algorithm. In: 15th International Conference on Software Reuse, ICSR 2016, Limassol, Cyprus (2016)
19. Font, J., Arcega, L., Haugen, Ø., Cetina, C.: Feature location in models through a genetic algorithm driven by information retrieval techniques. In: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, MODELS '16, pp. 272–282. ACM, New York, NY, USA (2016). https://doi.org/10.1145/2976767.2976789
20. Garca, S., Fernndez, A., Luengo, J., Herrera, F.: Advanced nonparametric tests for multiple comparisons in the design of experiments in computational intelligence and data mining: experimental analysis of power. Inf. Sci. **180**(10), 2044–2064 (2010). https://doi.org/10.1016/j.ins.2009.12.010
21. Gong, L., Lo, D., Jiang, L., Zhang, H.: Interactive fault localization leveraging simple user feedback. In: 2012 28th IEEE International Conference on Software Maintenance (ICSM), pp. 67–76 (2012). https://doi.org/10.1109/ICSM.2012.6405255
22. Grissom, R.J., Kim, J.J.: Effect Sizes for Research: A Broad Practical Approach. Earlbaum, Mahwah (2005)
23. Hassan, A.E., Holt, R.C.: The top ten list: dynamic fault prediction. In: 21st IEEE International Conference on Software Maintenance (ICSM'05), pp. 263–272 (2005). https://doi.org/10.1109/ICSM.2005.91
24. Haugen, Ø., Møller-Pedersen, B., Oldevik, J., Olsen, G.K., Svendsen, A.: Adding standardized variability to domain specific languages. In: Proceedings of the 2008 12th International Software Product Line Conference, SPLC '08, pp. 139–148. IEEE Computer Society, Washington, DC, USA (2008). https://doi.org/10.1109/SPLC.2008.25
25. Hoang, T.V., Oentaryo, R.J., Le, T.B., Lo, D.: Network-clustered multi-modal bug localization. IEEE Trans. Softw. Eng. (2018). https://doi.org/10.1109/TSE.2018.2810892
26. Holthusen, S., Wille, D., Legat, C., Beddig, S., Schaefer, I., Vogel-Heuser, B.: Family model mining for function block diagrams in automation software. In: Proceedings of the 18th International Software Product Line Conference: Volume 2, pp. 36–43 (2014). https://doi.org/10.1145/2647908.2655965
27. Kagdi, H., Gethers, M., Poshyvanyk, D., Hammad, M.: Assigning change requests to software developers. J. Softw. Evol. Process **24**(1), 3–33 (2012). https://doi.org/10.1002/smr.530
28. Kim, D., Tao, Y., Kim, S., Zeller, A.: Where should we fix this bug? A two-phase recommendation model. IEEE Trans. Softw. Eng. **39**(11), 1597–1610 (2013)
29. Kusumoto, S., Nishimatsu, A., Nishie, K., Inoue, K.: Experimental evaluation of program slicing for fault localization. Empir. Softw. Eng. **7**(1), 49–76 (2002). https://doi.org/10.1023/A:1014823126938
30. Lam, A.N., Nguyen, A.T., Nguyen, H.A., Nguyen, T.N.: Bug localization with combination of deep learning and information retrieval. In: 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC), pp. 218–229 (2017). https://doi.org/10.1109/ICPC.2017.24
31. Landauer, T.K., Foltz, P.W., Laham, D.: An introduction to latent semantic analysis. Discourse Process. **25**(2–3), 259–284 (1998). https://doi.org/10.1080/01638539809545028
32. Le, T.D.B., Oentaryo, R.J., Lo, D.: Information retrieval and spectrum based bug localization: Better together. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, pp. 579–590. ACM, New York, NY, USA (2015). https://doi.org/10.1145/2786805.2786880
33. Lehman, M.M., Ramil, J., Kahen, G.: A paradigm for the behavioural modelling of software processes using system dynamics. Tech. rep., Imperial College of Science, Technology and Medicine, Department of Computing (2001)
34. Liang, D., Harrold, M.J.: Equivalence analysis and its application in improving the efficiency of program slicing. ACM Trans. Softw.

Eng. Methodol. **11**(3), 347–383 (2002). https://doi.org/10.1145/567793.567796
35. Liu, D., Marcus, A., Poshyvanyk, D., Rajlich, V.: Feature location via information retrieval based filtering of a single scenario execution trace. In: Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering, ASE '07, pp. 234–243. ACM, New York, NY, USA (2007). https://doi.org/10.1145/1321631.1321667
36. Lopez-Herrejon, R.E., Linsbauer, L., Galindo, J.A., Parejo, J.A., Benavides, D., Segura, S., Egyed, A.: An assessment of search-based techniques for reverse engineering feature models. J. Syst. Softw. **103**, 353–369 (2015). https://doi.org/10.1016/j.jss.2014.10.037
37. Lopez-Herrejon, R.E., Linsbauer, L., Galindo, J.A., Parejo, J.A., Benavides, D., Segura, S., Egyed, A.: An assessment of search-based techniques for reverse engineering feature models. J. Syst. Softw. **103**, 353–369 (2015). https://doi.org/10.1016/j.jss.2014.10.037
38. Lukins, S.K., Kraft, N.A., Etzkorn, L.H.: Bug localization using latent dirichlet allocation. Inf. Softw. Technol. **52**(9), 972–990 (2010). https://doi.org/10.1016/j.infsof.2010.04.002
39. Mao, X., Lei, Y., Dai, Z., Qi, Y., Wang, C.: Slice-based statistical fault localization. J. Syst. Softw. **89**, 51–62 (2014). https://doi.org/10.1016/j.jss.2013.08.031
40. Marcus, A., Sergeyev, A., Rajlich, V., Maletic, J.: An information retrieval approach to concept location in source code. In: Proceedings of the 11th Working Conference on Reverse Engineering, pp. 214–223 (2004). https://doi.org/10.1109/WCRE.2004.10
41. Martinez, J., Ziadi, T., Bissyandé, T.F., Klein, J., l. Traon, Y.: Automating the extraction of model-based software product lines from model variants (t). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 396–406 (2015). https://doi.org/10.1109/ASE.2015.44
42. Martinez, J., Ziadi, T., Bissyandé, T.F., Klein, J., Traon, Y.L.: Bottom-up adoption of software product lines: a generic and extensible approach. In: Proceedings of the 19th International Conference on Software Product Line (SPLC), pp. 101–110 (2015). https://doi.org/10.1145/2791060.2791086
43. Neumann, G., Harman, M., Poulding, S.: Transformed Vargha-Delaney Effect Size, pp. 318–324. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22183-0_29
44. Panichella, A., Dit, B., Oliveto, R., Penta, M.D., Poshyvanyk, D., Lucia, A.D.: Parameterizing and assembling ir-based solutions for se tasks using genetic algorithms. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), vol. 1, pp. 314–325 (2016). https://doi.org/10.1109/SANER.2016.97
45. Poshyvanyk, D., Gueheneuc, Y.G., Marcus, A., Antoniol, G., Rajlich, V.: Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. IEEE Trans. Softw. Eng. **33**(6), 420–432 (2007). https://doi.org/10.1109/TSE.2007.1016
46. Powers, D.M.W.: Evaluation: from precision, recall and f-measure to roc., informedness, markedness & correlation. J. Mach. Learn. Technol. **2**(1), 37–63 (2011)
47. Rahman, M.M., Chakraborty, S., Ray, B.: Which similarity metric to use for software documents? A study on information retrieval based software engineering tasks. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE '18, pp. 335–336. ACM, New York, NY, USA (2018). https://doi.org/10.1145/3183440.3194997
48. Rao, S., Kak, A.: Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In: Proceedings of the 8th Working Conference on Mining Software Repositories, MSR '11, pp. 43–52. ACM, New York, NY, USA (2011). https://doi.org/10.1145/1985441.1985451
49. Revelle, M., Dit, B., Poshyvanyk, D.: Using data fusion and web mining to support feature location in software. In: IEEE 18th International Conference on Program Comprehension (ICPC), pp. 14–23 (2010). https://doi.org/10.1109/ICPC.2010.10
50. Saha, R.K., Lease, M., Khurshid, S., Perry, D.E.: Improving bug localization using structured information retrieval. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 345–355 (2013). https://doi.org/10.1109/ASE.2013.6693093
51. Salton, G., McGill, M.J.: Introduction to Modern Information Retrieval. McGraw-Hill Inc, New York (1986)
52. Salton, G., Wong, A., Yang, C.S.: A vector space model for automatic indexing. Commun. ACM **18**(11), 613–620 (1975). https://doi.org/10.1145/361219.361220
53. Sayyad, A.S., Ingram, J., Menzies, T., Ammar, H.: Scalable product line configuration: A straw to break the camel's back. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 465–474 (2013). https://doi.org/10.1109/ASE.2013.6693104
54. Sisman, B., Kak, A.C.: Incorporating version histories in information retrieval based bug localization. In: 2012 9th IEEE Working Conference on Mining Software Repositories (MSR), pp. 50–59 (2012). https://doi.org/10.1109/MSR.2012.6224299
55. Svendsen, A., Zhang, X., Lind-Tviberg, R., Fleurey, F., Haugen, Ø., Møller-Pedersen, B., Olsen, G.K.: Developing a software product line for train control: a case study of cvl. In: Proceedings of the 14th International Conference on Software Product Lines: Going Beyond, SPLC'10, pp. 106–120. Springer-Verlag, Berlin, Heidelberg (2010). http://dl.acm.org/citation.cfm?id=1885639.1885650
56. The English (porter2) Stemming Algorithm. http://snowball.tartarus.org/algorithms/english/stemmer.html (2002). Online; Accessed 04 April 2017
57. Thomas, S.W., Hassan, A.E., Blostein, D.: Mining Unstructured Software Repositories, pp. 139–162. Springer, Berlin, Heidelberg (2014). https://doi.org/10.1007/978-3-642-45398-4_5
58. Vargha, A., Delaney, H.D.: A critique and improvement of the cl common language effect size statistics of mcgraw and wong. J. Educ. Behav. Stat. **25**(2), 101–132 (2000). https://doi.org/10.3102/10769986025002101
59. Wang, S., Lo, D.: Amalgam+: composing rich information sources for accurate bug localization. J. Softw. Evol. Process **28**(10), 921–942 (2016). https://doi.org/10.1002/smr.1801
60. Wille, D., Holthusen, S., Schulze, S., Schaefer, I.: Interface variability in family model mining. In: Proceedings of the 17th International Software Product Line Conference: Co-located Workshops, pp. 44–51 (2013). https://doi.org/10.1145/2499777.2500708
61. Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F.: A survey on software fault localization. IEEE Trans. Softw. Eng. **42**(8), 707–740 (2016)
62. Zamani, S., Lee, S.P., Shokripour, R., Anvik, J.: A noun-based approach to feature location using time-aware term-weighting. Inf. Softw. Technol. **56**(8), 991–1011 (2014). https://doi.org/10.1016/j.infsof.2014.03.007
63. Zhang, X., Haugen, Ø., Moller-Pedersen, B.: Model comparison to synthesize a model-driven software product line. In: Proceedings of the 2011 15th International Software Product Line Conference (SPLC), pp. 90–99 (2011). https://doi.org/10.1109/SPLC.2011.24
64. Zhang, X., Haugen, Ø., Møller-Pedersen, B.: Augmenting product lines. In: Software Engineering Conference (APSEC), 2012 19th Asia-Pacific, vol. 1, pp. 766–771 (2012). https://doi.org/10.1109/APSEC.2012.76
65. Zhou, J., Zhang, H., Lo, D.: Where should the bugs be fixed? - more accurate information retrieval-based bug localization based on bug reports. In: Proceedings of the 34th International Conference on Software Engineering, ICSE '12, pp. 14–24. IEEE

Press, Piscataway, NJ, USA (2012). http://dl.acm.org/citation.cfm?id=2337223.2337226

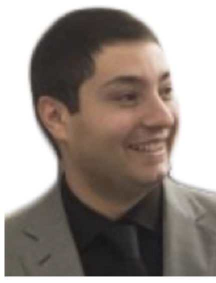
66. Zimmermann, T., Weisgerber, P., Diehl, S., Zeller, A.: Mining version histories to guide software changes. In: Proceedings of the 26th International Conference on Software Engineering, ICSE '04, pp. 563–572. IEEE Computer Society, Washington, DC, USA (2004). http://dl.acm.org/citation.cfm?id=998675.999460

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Lorena Arcega received the M.Sc. degree in advanced software technologies from Universidad San Jorge, Zaragoza, Spain. She is currently pursuing the PhD degree in computer science with the University of Oslo, Oslo, Norway. She is a Researcher with the SVIT Research Group, Universidad San Jorge. Her current research interests include software maintenance and evolution, variability modeling, and models at runtime.

Jaime Font received the PhD degree in computer science from University of Oslo, Oslo, Norway. He is an Assistant Professor with the SVIT Research Group, Universidad San Jorge, Zaragoza, Spain. His current research interests include reverse engineering, evolutionary computation, and variability modeling.

Øystein Haugen received the D.Sci. degree from the University of Oslo, Oslo, Norway. He is a Professor with Østfold University College, Halden, Norway, and a Senior Researcher with SINTEF ICT, Trondheim, Norway. His current research interests include cyber-physical systems, variability modeling and software product lines, object-oriented languages and methods, software engineering, and practical formal methods.



Carlos Cetina received the PhD degree in computer science from the Universitat Politècnica de València, Valencia, Spain. He is an Associate Professor with Universidad San Jorge, Zaragoza, Spain, and the Head of the SVIT Research Group. His current research interests include software product lines, variability modeling, and model-driven development.