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] Universidad San Jorge, SVIT Research Group, Zaragoza, Spain
{larcega,jfont,ccetina}@usj.es
[2] University of Oslo, Department of Informatics, Oslo, Norway
[3] Østfold University College, Department of Information Technology, 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).

Models at run-time provide a kind of formal basis for reasoning about the current system state, for reasoning about necessary adaptations, and for analyz-

ing 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].

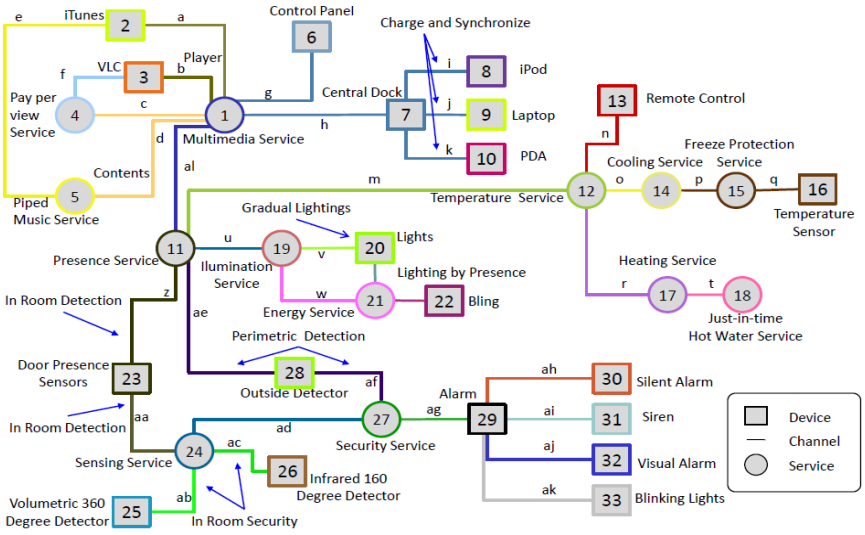

Fig. 1. Smart Hotel Architecture Model

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 Section 2, we present the Smart Hotel. In Section 3, we introduce our approach for feature location with architecture models at run-time. In Section 4, we evaluate our approach with the Smart Hotel and we discuss the results. In Section 5, we examine the related work of the area, and we present our conclusions in Section 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[4] is a DSL that describes pervasive systems using high-level abstraction concepts based on Meta-Object Facility (MOF) [1]. This language is

[4] https://tatami.dsic.upv.es/pervml/index.php
[1] Meta object facility (MOF) 2.0 core specification, 2003

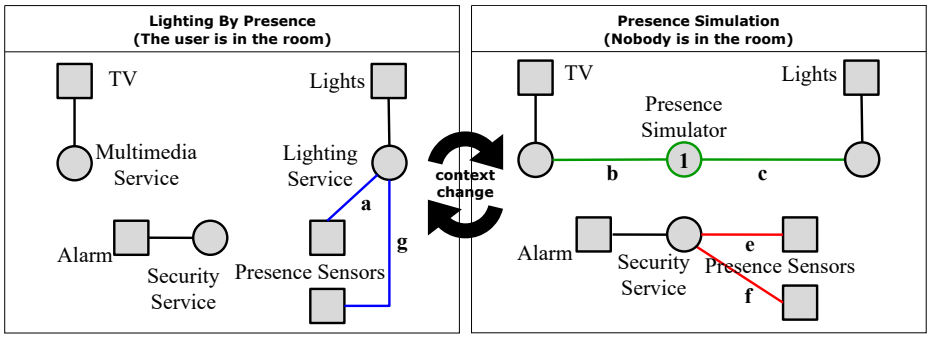

Fig. 2. Smart Hotel Architecture Model Reconfigurations

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).

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].

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 Figure 2 (right) shows a *Nobody in the room* configuration. As it can be

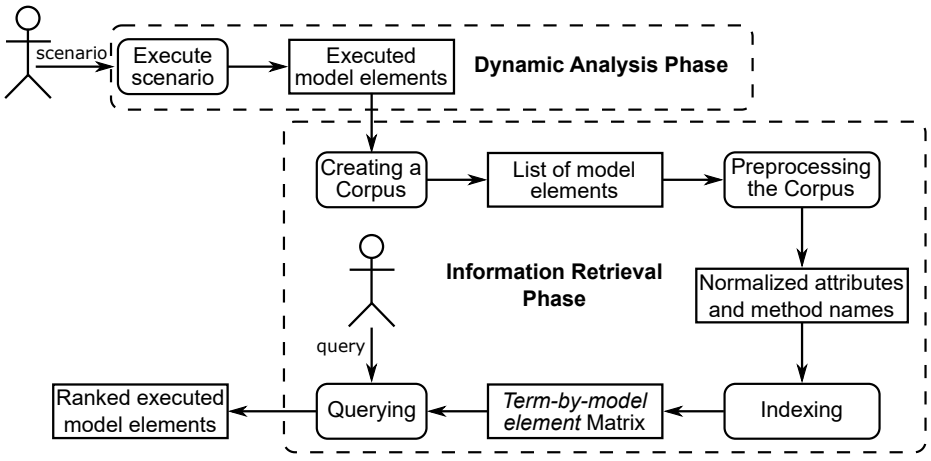

Fig. 3. Feature Location Approach based on Architecture Models at Run-Time

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

Fig. 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.

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 Section 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.'

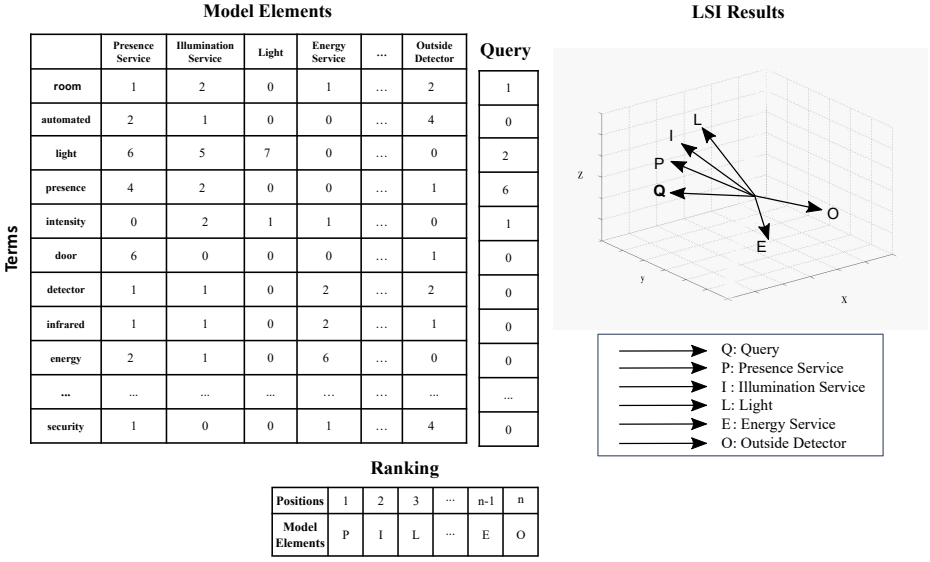

Fig. 4. Information Retrieval via Latent Semantic Indexing (LSI)

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 [18, 14, 17]. 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.

Fig. 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. Fig. 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 model elements of the run-time architecture model. Fig. 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:

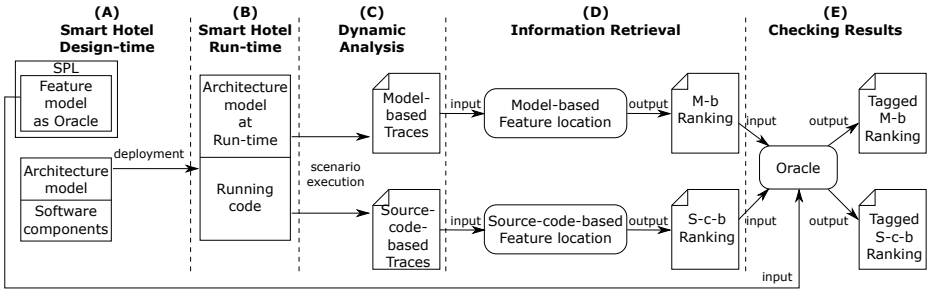

Fig. 5. The evaluation process followed in the Smart Hotel

- 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 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[5] 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 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

[5] KNX technology is a standard for applications in home and building control (http://www.knx.org/)

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.

Fig. 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.

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).

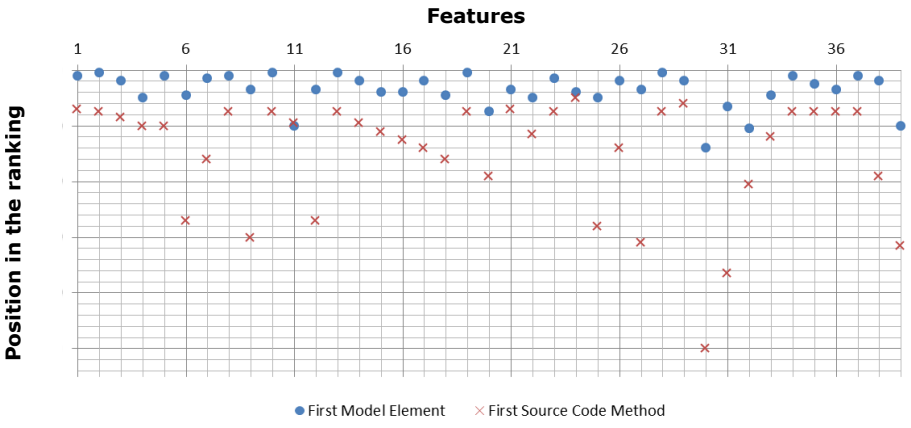

Fig. 6. Position of the first model element and the first method that belong to the target feature in the ranking for each one of the features

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.

Fig. 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 that have not been tagged have nothing at the end of the arrow. The angle corresponds to the cosine with which we calculated the position in the ranking (see Section 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.

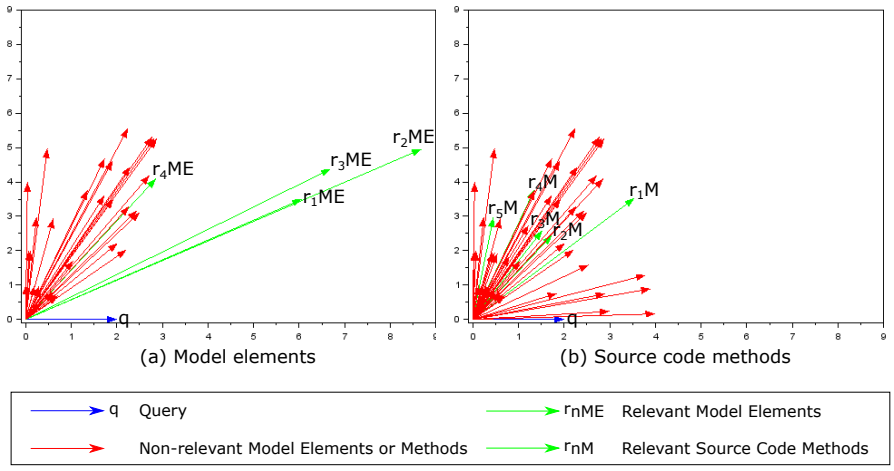

Fig. 7. Vectorial representation of the model elements and source code methods in the Ranking of the 'Gradual Lighting' feature

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 (May 2014), http://dx.doi.org/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. pp. 442–449. ICSE '96, IEEE Computer Society, Washington, DC, USA (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: Encyclopedia of Software Engineering. Wiley (1994)
4. Bencomo, N., Hallsteinsen, S., Santana de Almeida, E.: A view of the dynamic software product line landscape. Computer 45(10), 36–41 (Oct 2012)
5. Bencomo, N., France, R., Cheng, B.H.C., Aßmann, U. (eds.): Models@run.time. Foundations, Applications, and Roadmaps. Springer International Publishing (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.: Software Product Lines: Third International Conference, SPLC 2004, Boston, MA, USA, August 30-September 2, 2004. Proceedings, chap. Staged Configuration Using Feature Models, pp. 266–283. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
8. 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)
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 '15, 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: 15th International Conference on Software Reuse. ICSR 2016, Limassol, Cyprus (Jun 2016)
11. Koschke, R., Quante, J.: On dynamic feature location. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering. pp. 86–95. ASE '05, ACM, New York, NY, USA (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 processes 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. Tech. rep., Imperial College of Science, Technology and Medicine, Department of Computing (Sep 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. pp. 234–243. ASE '07, ACM, New York, NY, USA (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 '15, 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 Transactions on Software Engineering 33(6), 420–432 (Jun 2007), http://dx.doi.org/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: Program Comprehension (ICPC), 2010 IEEE 18th International Conference on. pp. 14–23 (June 2010)
19. Revelle, M., Poshyvanyk, D.: An exploratory study on assessing feature location techniques. In: Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on. 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 Heidelberg (2013)
21. Travassos, M.O.B.G.H.: Contributions of in virtuo and in silico experiments for the future of empirical studies in software engineering. In: 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 (Oct 2012)