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] 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. 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 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 [21, 16, 20]. 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 [23, 17]. 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 Section 2, we present the Smart Hotel and the model traces. In Section 3, we describe our approach for Dynamic Feature Location with models. In Section 4, we evaluate our approach in the Smart Hotel and we discuss the results. In Section 5, we examine the related work of the area. Finally, we present our conclusions in Section 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 [13, 6]. For more details about the reconfiguration engine of the Smart Hotel see [7].

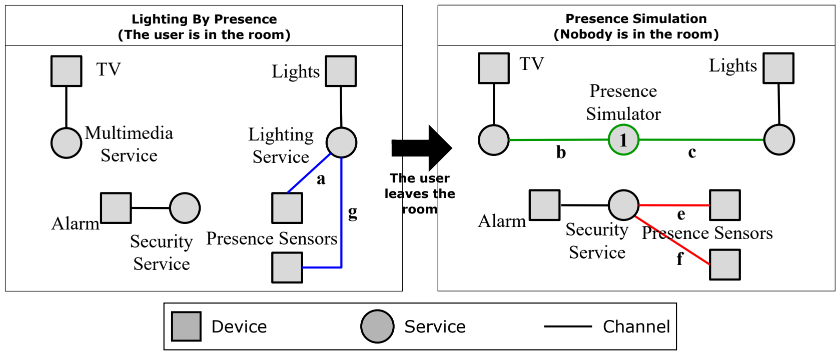


Fig. 1. Smart Hotel Model Reconfigurations

Fig. 1 shows two Smart Hotel configurations according to the concrete syntax of the architecture model of PervML [19]. Fig. 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 or created even if the model does not correspond to a target configuration of the system.

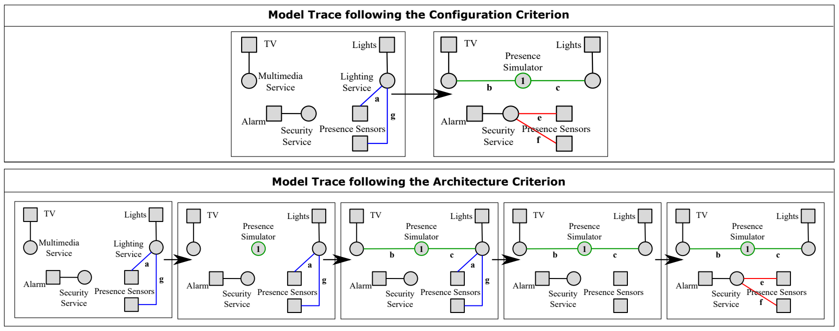


Fig. 2. Different Model Traces following the different Criterion

Fig. 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

Fig. 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.

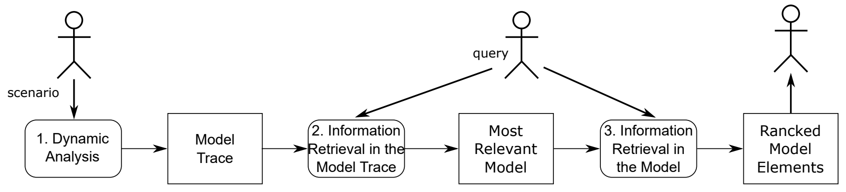


Fig. 3. Overview of the Dynamic Feature Location Approach

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 Section 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) [21, 16]. 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.

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

Models

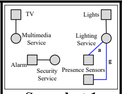
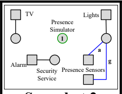
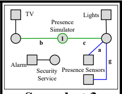
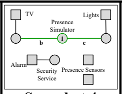
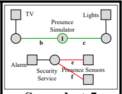
Terms	 Snapshot 1	 Snapshot 2	 Snapshot 3	 Snapshot 4	 Snapshot 5	Query
room	1	2	0	1	2	1
automated	2	1	0	0	4	0
light	6	5	7	0	0	2
presence	4	2	0	0	1	6
intensity	0	2	1	1	0	1
...	...	...	...	...	...	...
security	1	0	0	1	4	0

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

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. Fig. 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. 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 models of the model trace. Fig. 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

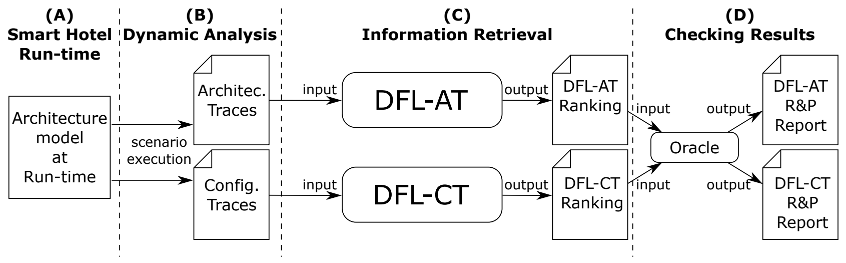


Fig. 5. Overview of the Evaluation process

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 [23, 17]. 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.

Fig. 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 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).

Fig. 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.

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

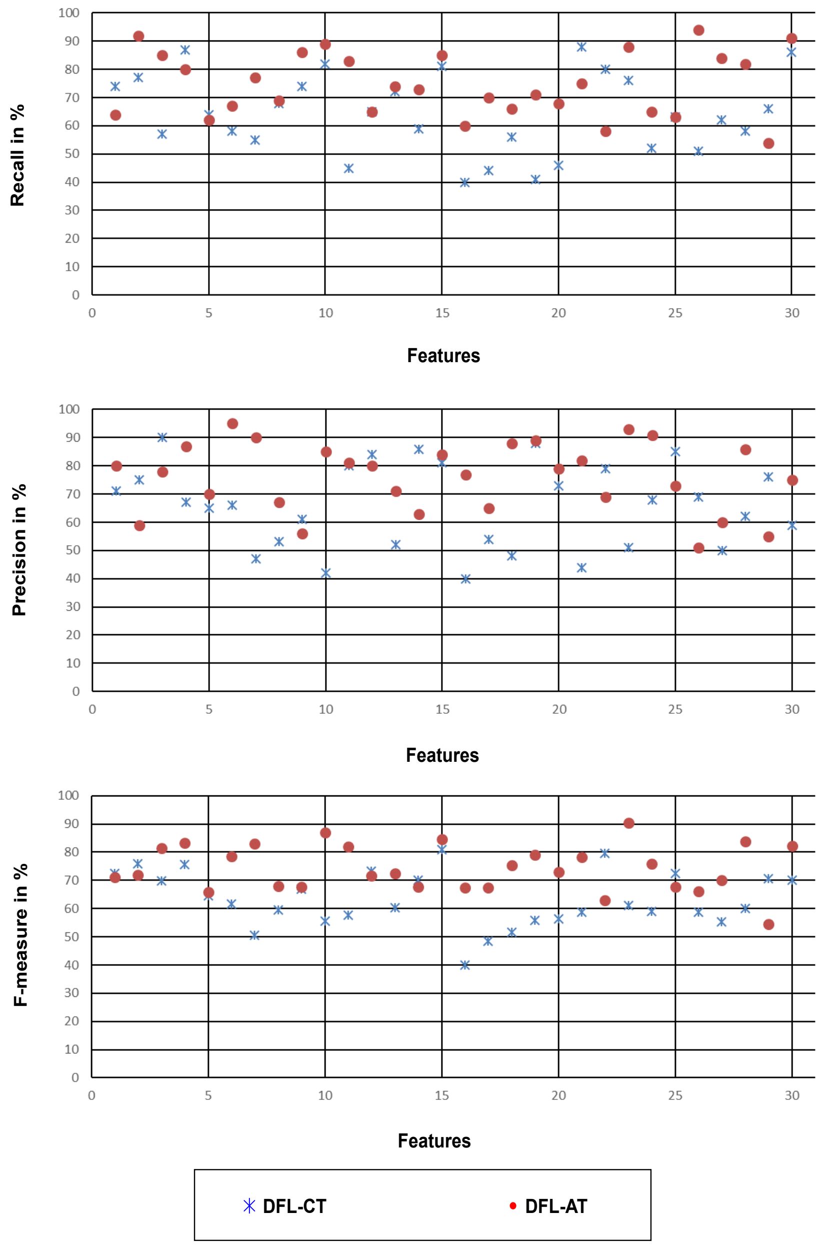


Fig. 6. Recall, Precision and F-Measure Graphs

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 Section 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 features 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 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. 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.
2. G. Antoniol and Y.-G. Gueheneuc. Feature identification: An epidemiological metaphor. *IEEE Trans. Softw. Eng.*, 32(9):627–641, Sept. 2006.
3. L. Arcega, J. Font, Ø. Haugen, and C. Cetina. Feature location through the combination of run-time architecture models and information retrieval. In J. Grabowski and S. 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.
4. V. R. Basili. The role of experimentation in software engineering: Past, current, and future. In *Proceedings of the 18th International Conference on Software Engineering*, ICSE '96, pages 442–449, Washington, DC, USA, 1996. IEEE Computer Society.
5. V. R. Basili, G. Caldiera, and H. D. Rombach. The goal question metric approach. In *Encyclopedia of Software Engineering*. Wiley, 1994.
6. 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.
7. C. Cetina. *Achieving Autonomic Computing through the Use of Variability Models at Run-time*. PhD thesis, Universidad Politécnica de Valencia, 2010.
8. 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.
9. B. Dit, M. Revelle, and D. Poshyvanyk. Integrating information retrieval, execution and link analysis algorithms to improve feature location in software. *Empirical Software Engineering*, 18(2):277–309, 2013.
10. J. Font, L. Arcega, Ø. Haugen, and C. Cetina. 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.
11. J. Font, L. Arcega, Ø. Haugen, and C. Cetina. Feature location in model-based software product lines through a genetic algorithm. In *15th International Conference on Software Reuse*, ICSR 2016, Limassol, Cyprus, Jun 2016.
12. A. 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.
13. IBM. An architectural blueprint for autonomic computing. Technical report, IBM, 2006.
14. T. K. Landauer, P. W. Foltz, and D. Laham. An introduction to latent semantic analysis. *Discourse Processes*, 25(2-3):259–284, 1998.
15. M. M. Lehman, J. Ramil, and G. 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.
16. D. Liu, A. Marcus, D. Poshyvanyk, and V. 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.

17. A. Marcus, A. Sergeyev, V. Rajlich, and J. 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.
18. J. Martinez, T. Ziadi, T. F. Bissyandé, and Y. 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.
19. J. Muñoz. *Model Driven Development of Pervasive Systems. Building a Software Factory.* PhD thesis, Universidad Politécnica de Valencia, 2008.
20. D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, and V. 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.
21. 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.
22. H. E. Salman, A. Seriai, and C. 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.
23. G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill, Inc., New York, NY, USA, 1986.
24. 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.