

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 Cetina¹

¹ San Jorge University, SVIT Research Group, Zaragoza, Spain
{larcega,jfont,ccetina}@usj.es

² University of Oslo, Department of Informatics, Oslo, Norway
oystein.haugen@ifi.ui.no

³ Østfold University College, Department of Information Technology, Halden, Norway
oystein.haugen@hiiof.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

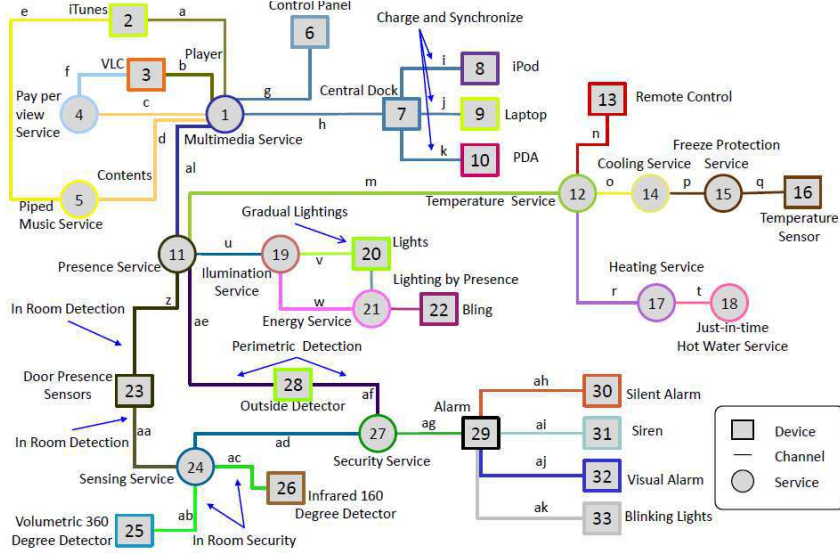


Fig. 1. Smart Hotel Architecture Model

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

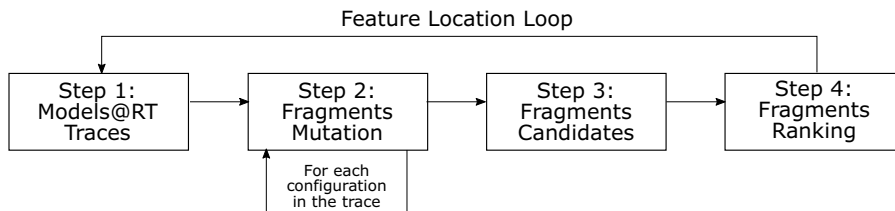


Fig. 2. Feature Location Algorithm

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.

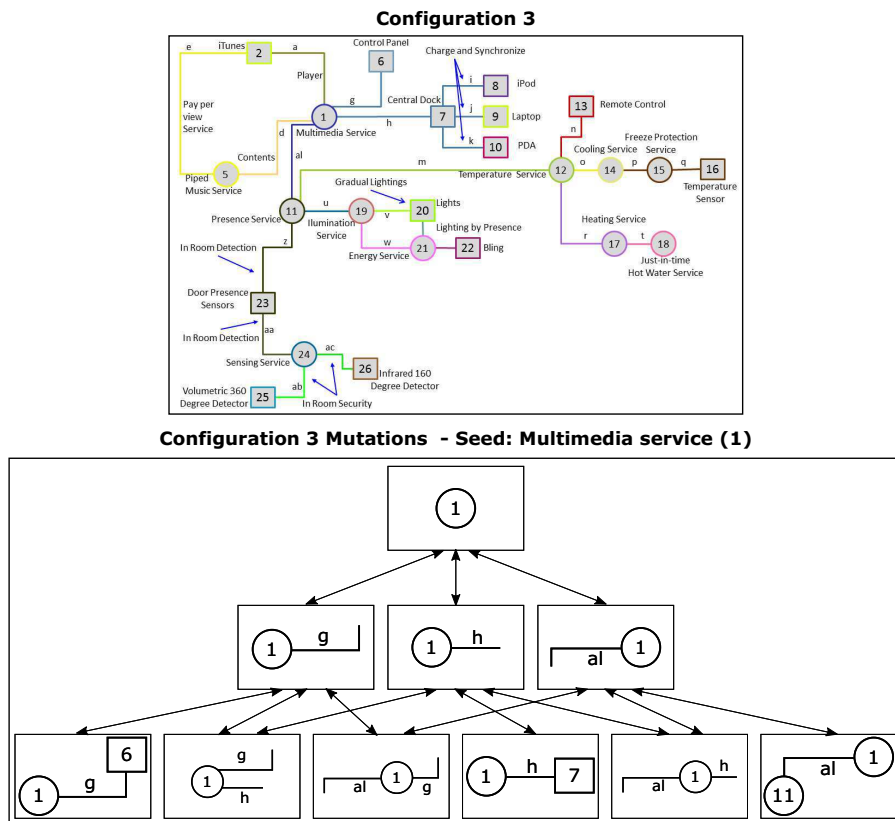


Fig. 3. Configuration 3 and Mutations of the Seed Fragment

We executed the Smart Hotel software during a time. The Smart Hotel re-configured 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

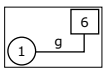
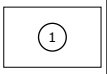
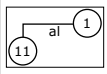
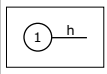
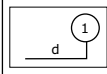
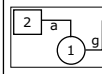
Model fragments			...			...		
Occurrences in configurations	10/12	10/12	...	8/12	7/12	...	3/12	1/12
Occurrences in reconfigurations	25/30	25/30	...	19/30	19/30	...	9/14	4/14

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).

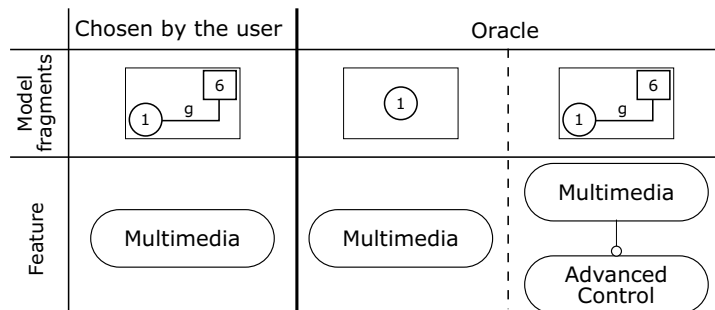


Fig. 5. Results of the Evaluation

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)