Identification Of Variation Points Using Dynamic Analysis

Bas Cornelissen
Delft University of Technology
The Netherlands
s.g.m.cornelissen@ewi.tudelft.nl

Bas Graaf
Delft University of Technology
The Netherlands
b.s.graaf@ewi.tudelft.nl

Leon Moonen
Delft University of Technology and CWI
The Netherlands
Leon.Moonen@computer.org

Abstract

In this position paper we investigate the use of dynamic analysis to determine commonalities and variation points as a first step to the migration of similar but separate versions of a software system into an integrated product line. The approach detects forks and merges in different execution traces as an indication of variation points. It is illustrated by a simple implementation, which is applied to an academic example. Finally we formulate a number of research issues that need to be investigated further.

1. Introduction

Already many successes have been reported with respect to the use of product line approaches in software development organizations [1]. A company that migrates to a product line approach must define a product line architecture that incorporates the design decisions common to all product line members. Additionally, the variability between the different product line members is to be made as explicit as possible.

In practice, the idea of following a product line approach can be applied in various levels of detail. For example, one can define a reference architecture which specifies all commonalities between products but does not make the variation points explicit. As such, we can distinguish between various maturity levels in a product line deployment [2]. This is also illustrated in an industrial example discussed by Graaf et al. [3].

A typical situation in which the adoption of more product line concepts, and thereby raising the maturity level, is beneficial, is when a company has developed several versions of a product for different customers. All these versions are extended in one or more ways with respect to some original system that was initially developed. At some point a customer comes along that requires some of the extensions that were already implemented, but for different versions of the product, and thus their implementations reside in different development branches. As more versions are being developed, such a situation becomes more and more likely. At that point these extensions should be reengineered into clearly defined, configurable features by making *variation points* explicit, ideally enabling late binding.

Domain and application engineering methods have been proposed to solve this problem. Typically these approaches are applied in a context where a product line is developed from scratch, and do not take existing source code into account. However, new product lines are typically not developed from scratch, but evolve from a set of similar, traditionally developed products. Furthermore, often many design decisions are only explicit in the source code.

In this paper we consider the problem of detecting forks and merges in the execution traces generated by different versions of a system so as to identify its variation points.

The remainder of this paper is organized as follows. Section 2 discusses some related work. In Section 3 the basic idea of how execution traces can help in identifying variation points is presented. Section 4 explains a simple implementation of this idea that detects forks and merges in execution traces. This implementation is applied to a simple example in Section 5. The paper is concluded with some discussions and directions for future work in Section 6.

2. Related Work

Van Gurp et al. [4] provide an excellent introduction to the concepts of variability in software product lines and discuss how variability can be documented using feature graphs. However, they do not discuss in much detail how commonalities and variation points can be discovered.

Approaches for domain engineering aim at identifying commonalities and variabilities for the definition of product line architectures. Scope, variability, and commonality (SCV) analysis discussed by Coplien et al. [5] provides a systematic way of thinking about commonality and variability. The same work also introduces FAST, an approach for domain engineering based on SCV-thinking. Other domain engineering approaches are FODA [6] and

FORM [7]. Typically these approaches are based on the analysis of high-level information, such as requirements to identify variabilities and commonalities.

Execution traces have been used for many purposes in the program analysis community. However, in only a few cases traces from different programs were compared to each other. Much of the work is concerned with identifying which components are required for a specific feature or set of features.

The software reconnaissance technique proposed by Wilde and Scully [8] compares execution traces of different sets of scenarios to identify which components are required for a specific feature.

Eisenbarth et al. [9] apply formal concept analysis to execution traces that each exhibit a different feature, so as to identify feature-component relations. As such they also investigate the commonalities and variabilities between different features in terms of the components required to implement them.

These approaches compare different execution traces of the same program. Therefore, they rely on the assumption that the exhibition of a certain feature can be controlled by the user, which is not always the case.

3. Tracing and variation points

Suppose we have two branches of a software system, one being the base system and the other a variant with one or more additional features. Detection of variation points using execution traces is based on the idea illustrated by Figure 1. In this graph, we have projected one trace on top of the other. Each node in the graph denotes the usage of a component for the execution of a scenario. The arcs indicate the order in which the components were used. The fork in Figure 1a can be considered the variation point. All behavior executed up to the split is common behavior and the components that are used after the split are feature-specific.

The components considered in an execution trace are units of source code. Different levels of granularity are possible: statement, method, class, package or other abstractions.

Execution traces are obtained by executing some scenario. Comparison of execution traces is only meaningful when the corresponding scenarios are either the same or very similar. In this context a scenario is defined by the input offered to the system. We do not consider the system's response as part of the scenario, as the intention is to execute scenarios on different systems that yield different responses.

For the localization of variation points in the implementation that correspond with the specific features, we need two execution traces: one in which the extension is exhibited and one in which it is not. Depending on the feature, it may or may not be possible for the two scenarios that generate these traces to be identical. In case the exhibition of a certain feature depends on the input, different scenarios are needed. This can be the case, for example, when the feature is only activated when a user clicks a certain GUI button. If activation does not depend on the scenario, we compare execution traces generated by various versions of a system.

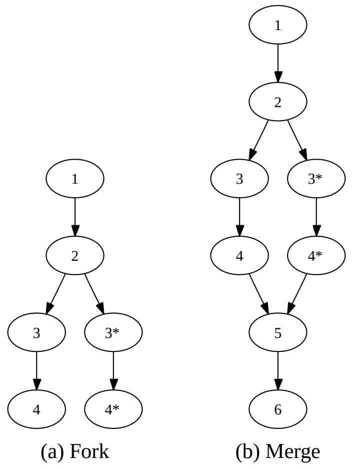

Figure 1: Forks and merges in an execution trace

The underlying assumption in our approach is that both execution traces will largely resemble each other and the associated graphs will have most nodes in common, up to the point where the additional feature is exhibited (Figure 1a). Automatic detection of such a fork is trivial: we take the node before the first deviation in the two execution traces. The detection of a merge, however, is more involved. Simply detecting the first pair of nodes that are identical after the fork might not be meaningful. Usage of a specific component in both traces does not necessarily imply that the same behavior was demonstrated from a user's perspective. The next Section will describe a solution to this issue using an evolving comparison window.

The generation of traces that can be compared meaningfully is even more complicated if non-deterministic behavior is considered (e.g. in games).

4. Approach

In this section, we first present the running example that is used in the remainder of this paper to illustrate our approach for the identification of variation points using execution traces. Next, we explain how we obtain those traces and finally how they are processed.

4.1. Running example: Pacman

The system we use as a running example in this paper is a java-based game called Pacman. With a little imagination, we can regard Pacman as a simple example of a software product line.

Pacman is a modest software system consisting of 20 java classes and approximately 1000 lines of code. Like in a software product line there exist several variants of this system, each with distinct added features.

For example, in the reference system there is one hard-coded map being loaded whenever a game is played. In another version, which can be considered a member in our product line, functionality has been added (in a separate class) to read user-defined maps from a file. Yet another version of the system features an additional type of entities on the map with which the player and the monsters can interact.

4.2. Dynamic analysis using aspects

We obtain execution traces by instrumenting the system with trace statements. We add these trace statements by means of aspect-oriented programming. Aspect-oriented programming is extremely suitable for implementing a crosscutting concern such as tracing since it allows us to add code at various program locations with limited effort. We use AspectJ to weave additional code in the system such that, whenever a method is called, a message is printed to a log file. This message contains both the method being called and the class to which this method belongs.

Now, we can generate traces containing the methods called during execution. Depending on the desired level of granularity of variation point detection, we may need to further process this trace, e.g. to generate a trace on the class level.

Alternatively, one could use the Java Virtual Machine Profiler Interface (JVMPI) to collect traces from a system, as is done, for example, by Reiss and Renieris [10].

4.3. Determining variation points

When dealing with software product lines, each of the product line members generally contains a set of features. Typically, the members have some of these features in common whereas others are product-specific. If an architect is to combine two or more products, the components responsible for the latter type of features - the *variation points* - must be determined.

We propose a method in which we compare the traces generated by two versions of a similar system to discover variation points. On the one hand we have a trace generated by the reference system, called the *reference trace*, and on the other hand a trace generated by an extended version, called the *feature trace*. These traces are to be obtained by running both systems using similar scenarios: ideally, the latter differs from the former only in that the specific extension is exhibited.

As mentioned in section 3, branches are not necessarily considered merged as soon as the two traces once again have one method in common. For this reason, we require the traces to have *multiple* consecutive methods in common.

The algorithm being applied reads as follows:

1. Compare the traces of the reference system and the product line member line by line.

2. If the two methods at hand differ, the traces have **split** into branches. Create an *N*-size checksum of the current reference method and the next *N-1* methods (henceforth, we will call this the *reference window*).

3. Next, create a checksum of the upcoming N methods in the feature trace, thus creating the *feature window*.

4. If the checksums are equal, the branches are considered to have **merged**. If they do not match, shift the feature window down one method, thus creating a new feature checksum. Repeat this step a maximum of *M* times.

5. If the checksums still do not match, shift the reference window down one method. Repeat the previous step, but repeat the current step a maximum of *M* times.

6. If there is still no match, either *M* is too small or the branches never merge, i.e. the systems never again exhibit the same behavior at the method level.

The values for *N* and *M* are variable and depend on several factors. In assigning suitable values to these variables, important factors include the size of the system and the predicted impact (in terms of the amount of associated method calls) of the feature at hand. We expect the architect to have sufficient knowledge of the system at hand to choose appropriate values for *M* and *N*.

The branching behavior derived by the algorithm can be visualized by presenting contexts (of predefined sizes) of all forking and merging points in the traces to the user. By visualizing and inspecting the branching behavior, the architect has a way of identifying which methods and classes account for member-specific features. Having approximated these variation points, it takes much less effort to merge the two versions than if the entire systems had required close inspection.

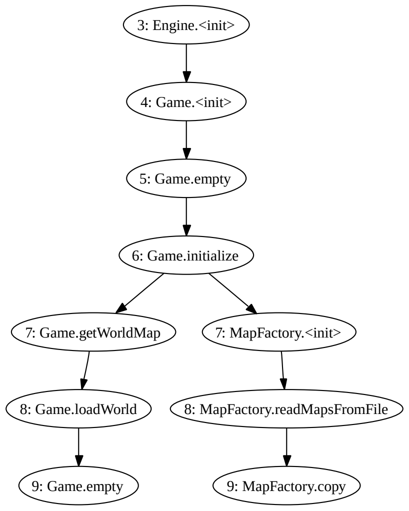

Figure 2: A fork and its context in the trace.

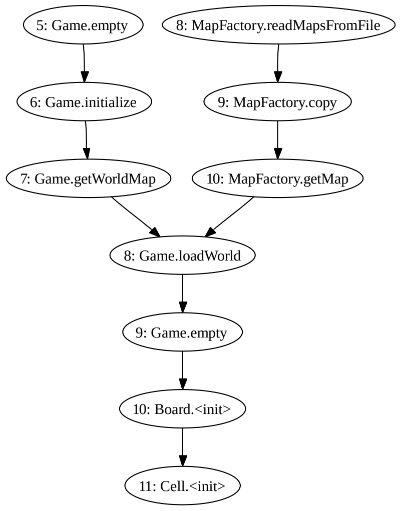

Figure 3: A merge and its context in the trace.

5. Preliminary results

To illustrate the method presented in the previous section we have conducted some experiments on the Pacman system described earlier.

In this section, we will highlight the experiment involving Pacman's reference system and the modified version featuring separate map handling.

5.1. Generating traces

Choosing appropriate scenarios is relatively easy in this case, as loading maps is part of the initialization phase and therefore not subject to human intervention. It is simply a matter of running both programs and exiting without actually having played game.

Part of a method trace as generated by use of the aspect mentioned in section 4.2 is depicted in Listing 1.

```
Pacman.main
Pacman.<init>
Engine.<init>
Game.<init>
Game.empty
Game.empty
Game.<init>
Game.initialize
...
```

Listing 1: Part of a method trace.

Incorporation of the actual stack depths is not part of the results discussed here and is subject to future research.

5.2. Branching behavior

We are now ready to compare the traces by using the algorithm described in section 4.3. However, we need to define some parameters first.

Since we are considering a small system and a not so complicated feature, we do not expect branches to be very long, e.g. perhaps tens of methods at most. For the same reason we will set the checksum size at a relatively small value, e.g., 5 methods. Finally, the size of the context being presented to the user is set to 7.

The results can be viewed in figures 2 and 3. Figure 2 depicts the context of the point where the feature trace started deviating from the reference trace. One can easily see that whereas in the original version a local method is invoked to get a map, the other version instantiates a whole new class that deals with the map handling.

Figure 3 illustrates that not many methods calls later, the branches have merged. From here on, the traces are apparently similar.

Judging by the visualizations - if presented at the correct abstraction level - an architect can easily isolate the feature-specific part and, if desired, migrate the components associated with this variation point towards other existing product line members.

6. Discussion and Future Work

Effort To repeat our experiment on a different subject system, one can apply the following process:

1. Perform a quick (1 hour) exploration of the system to gain some insight in its structure. This provides an initial estimate for the values of the *M* and *N* parameters.

2. Determine appropriate scenario(s) that exercise the

desired features.

3. Add tracing instrumentation to the system, e.g. by weaving aspects.

4. Collect execution traces for given scenarios and (automatically) compare them to find variation points.

5. If desired, repeat step 4 using alternative values for *M* and *N* to fine-tune the results.

Precision In the current implementation we more or less assume that a merge point is not located arbitrarily far from a fork. Hence, we introduced the *M*-parameter in our detection algorithm. This assumption is valid because we require that one version is a strict extension over the other.

If we abandon this requirement, we would have to search *both* execution traces all the way to the end to find potential merges. The complexity of this search is $O(n^2)$, which could be problematic for systems of realistic size, involving traces consisting of millions of components. This is why we advocate a sensible value for *M*: a value defined by the architect, based on how much impact he expects the particular feature to have on the given trace granularity level (method level in the case presented here). In the future we may be able to automatically determine optimal values for specific systems.

Future work To render identification of variation points feasible in the case of complex systems, we need more refined techniques. One approach is to take into account not only the methods being called but also their actual parameters. This would require a straightforward extension of the tracing instrumentation.

Another option is to also look at the stack depth or maybe even the complete stack whenever a method is called. Both these extensions to our technique potentially allow for the detection of extra forks, and increase the probability that an identical entry in the two call traces indeed implies that the two versions were again exhibiting common behavior, from a user's perspective. Probably this also means that the *N*-parameter can be smaller, which in turn reduces the cost of the checksum calculations.

An alternative approach in dealing with systems of realistic size would be to not directly analyze the method trace, but to first lift its elements to higher levels of abstraction, e.g. from methods to classes or packages. To this end, we would first have to extract information with respect to the structural decomposition of the system.

Finally, once a feature is localized a next step is to modularize the code that implements it. To provide guidelines for this step we will investigate whether the number of times two traces intersect (in terms of identical methods being called) before the same behavior is exhibited (as defined by the *N*-parameter) could be a measure for the degree of 'crosscuttingness' of a feature, and hence for the effort required to reengineer such a feature into a reusable component.

Acknowledgements

This work was partially supported by NWO Jacquard project Reconstructor.

References

[1] Software Engineering Institute. Product Line Hall of Fame. `http://www.sei.cmu.edu/productlines/plp_hof.html`, 2005 .

[2] Jan Bosch. Maturity and evolution in software product lines: Approaches, artefacts and organization. In *Proceedings of the Second International Conference on Software Product Lines (SPLC 2)*. Springer-Verlag, August 2002.

[3] Bas Graaf, Hylke van Dijk, and Arie van Deursen. Evaluating an embedded software reference architecture – industrial experience report. In *Proceedings of the 9th European Conference on Software Maintenance and Reengineering (CSMR)*, Manchester, UK, March 21-23 2005. IEEE Computer Society.

[4] Jiles van Gurp, Jan Bosch, and Mikael Svahnberg. On the notion of variability in software product lines. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture(WICSA'01)*. IEEE Computer Society, August 2001.

[5] James Coplien, Daniel Hoffman, and David Weiss. Commonality and variability in software engineering. *IEEE Software*, 15(6):37–45, November 1998.

[6] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, 1990.

[7] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. FORM: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5:143–168, January 1998.

[8] N. Wilde and M.C. Scully. Software reconnaissance: Mapping program features to code. *Journal on Software Maintenance: Research and Practice*, 7(1):49–62, January 1995.

[9] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Feature-driven program understanding using concept analysis of execution traces. In *Proceedings of the Ninth International Workshop on Program Comprehension (IWPC'01)*. IEEE Computer Society, May 2001.

[10] Steven P. Reiss and Manos Renieris. Generating Java trace data. In *Proceedings of the ACM 2000 conference on Java Grande*. ACM Press, 2000. ISBN 1-58113-288-3.