Locating Distinguishing Features Using Diff Sets

Julia Rubin
University of Toronto, Canada
and
IBM Research at Haifa, Israel
mjulia@il.ibm.com

Marsha Chechik
University of Toronto, Canada
chechik@cs.toronto.edu

ABSTRACT

In this paper, we focus on the problem of feature location for families of related software products realized via code cloning. Locating code that corresponds to features in such families is an important task in many software development activities, such as support for sharing features between different products of the family or refactoring the code into product line representations that eliminate duplications and facilitate reuse. We suggest two heuristics for improving the accuracy of existing feature location techniques when locating *distinguishing* features – those that are present in one product variant while absent in another. Our heuristics are based on identifying code regions that have a high potential to implement a feature of interest. We refer to these regions as *diff sets* and compute them by comparing product variants to each other. We exemplify our approach on a small but realistic example and describe initial evaluation results.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*; D.2.13 [**Software Engineering**]: Reusable Software—*Reuse Models*

General Terms

Design, Algorithms

Keywords

Software product lines, software maintenance, feature location.

1. INTRODUCTION

Software Product Line Engineering (SPLE) [2] is an engineering discipline supporting efficient development and maintenance of related software products. It capitalizes on identifying and managing *common* and *variable* product line features across a product portfolio and promotes *systematic* software reuse by leveraging the knowledge about the set of available features, relationships among the features, and traceability between the features and software artifacts that implement them.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ASE '12, September 3-7, 2012, Essen, Germany
Copyright 2012 ACM 978-1-4503-1204-2/12/09 ...$15.00.

While SPLE promotes systematic reuse of features between product variants, in reality product families often emerge ad-hoc, when companies have to release a new product that is similar, yet not identical, to existing ones. In many cases, new products are created using code cloning mechanisms (the "clone-and-own" approach) when an existing product is copied and later modified independently from the original version [6, 11, 4].

A feature implemented in one cloned variant might often be useful for another. Thus, its code should be *located* and *copied* to that different variant, promoting sharing of features between products [17]. Moreover, numerous approaches, e.g., [10, 1], advocate refactoring of cloned program variants into "single-copy" representations, eliminating duplications and explicating variabilities (e.g., the *annotative* or *compositional* SPLE approaches [8]). Identifying traceability between product features and artifacts that realize those features can help with such refactorings: in many cases, the set of features is known upfront and is specified by the product documentation whereas the relationship between the features and their corresponding implementation is rarely documented.

Feature location techniques aim at solving this problem by locating pieces of code that implement a specific program functionality, a.k.a. a *feature*. Numerous feature location approaches that are based on static program analysis, dynamic analysis, information retrieval (IR) techniques, change set analysis, or a combination of several aforementioned techniques are extensively studied in the literature [3, 16]. Obviously, each of the existing feature location techniques can be used for locating features of products in a product family when treating these related products as singular independent entities. However, leveraging any available information that can help improve accuracy of existing feature location techniques can be beneficial, as the accuracy of many contemporary techniques is still low (for example, see [13]).

In this paper, we propose to extend the set of heuristics used by feature location techniques when locating *distinguishing* features – those that are present in one but not all variants of a product family realized via code cloning. Our heuristics are based on additional information available when considering *multiple product variants together*. Such information can be obtained by comparing the code of a variant that contains a particular feature of interest to the one that does not. The comparison provides an initial "coarse-grained partitioning" of a program into relevant and irrelevant parts and assists the feature location process: the features of interest are implemented in the *unshared* parts of the program, providing a clue where to locate them. We thus detect and explicitly capture information about the *unshared* parts of the program as a separate artifact, called a *diff set*, and propose heuristics that use this artifact for improving the accuracy of feature location.

The remainder of this paper is structured as follows: In Section 2, we illustrate and motivate the problem. In Section 3, we define *diff sets* and outline the two heuristics based on them. Our preliminary results are reported in Section 4. Section 5 discusses related work

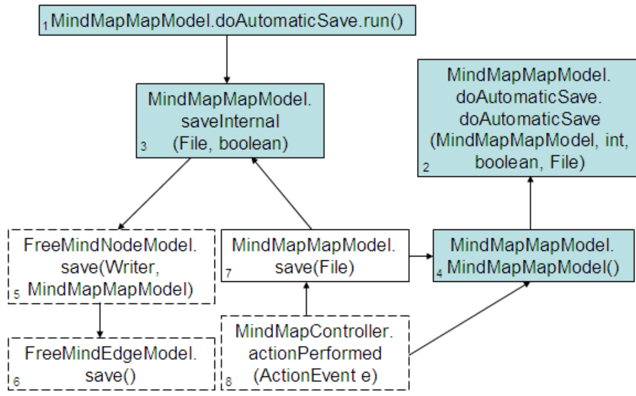

Figure 1: A snippet of the *automatic save File* call graph.

```
public MindMapMapModel( MindMapNodeModel root,
    ⋮                      FreeMindMain frame ) {
  // automatic save:
  timerForAutomaticSaving = new Timer();
  int delay = Integer.parseInt(getFrame().
    getProperty("time_for_automatic_save"));
  int numberOfTempFiles = Integer.parseInt(getFrame().
    getProperty("number_of_different_files_for_automatic_save"));
  boolean filesShouldBeDeletedAfterShutdown = Tools.
    safeEquals(getFrame().
    getProperty("delete_automatic_save_at_exit"),"true");
  String path = getFrame().getProperty("path_to_automatic_saves");
    ⋮
  timerForAutomaticSaving.schedule(new doAutomaticSave(
    this, numberOfTempFiles,
    filesShouldBeDeletedAfterShutdown, dirToStore),
    delay, delay);
  );
}
```

Figure 2: The *automatic save File* code snippet.

in the area. We present possible directions of future work and conclude the paper in Section 6.

2. EXAMPLE

Consider the problem of locating the *automatic save file* feature in the code of the Freemind open source mind-mapping tool[1], previously studied in [15]. Figure 1 presents the relevant Freemind's call graph snippet, where the four methods which contribute to the feature implementation are shaded (elements #1-4). Central to the implementation of the feature is the `run` method of the `MindMapMapModel`'s sub-class `doAutomaticSave` (element #1) which calls the `saveInternal` method (element #3), responsible for performing the *save* operation itself. The `doAutomaticSave` class is initiated by the `MindMapMapModel`'s constructor (element #4), as shown in Figure 2. The constructor assigns values to several configuration parameters related to the *automatic save file* function, initializes the `doAutomaticSave` class by calling its constructor (element #2) and then registers the class on the scheduling queue. The scheduler (not shown in Figure 1) calls the `run` methods of all elements added to its queue, including the `run` method of `doAutomaticSave` (element #1).

The implementation of the *automatic save file* feature is integrated into the Freemind code and relies on additional program elements. For example, element #3 also initiates a call to method `save(Writer, MindMapMapModel)` of `FreeMindNodeModel` (element #5), which, in turn, calls element #6 – `save(Writer, MindMapMapModel)`. Element #3 itself is called by `MindMap-MapModel`'s `save(File)` (element #7), which is called by `MindMap-Controller`'s `actionPerformed(ActionEvent)` (element #8). These methods are not relevant to the feature implementation because they handle a *user-triggered save* operation instead of *automatic save*. In fact, element #8 initiates calls to additional 24 methods, all of which are irrelevant to the implementation of the feature. In Figure 1, irrelevant methods are not shaded.

Comparing the variant of the Freemind software that has the *automatic save file* feature to another one, that does not have that feature, produces a set of *unshared* elements that distinguish between the variants – the *diff set*. In Figure 1, unshared elements are denoted by solid-line boxes while common ones have dotted-line boxes. Since our goal is to detect the feature that is present in one but absent in another variant of the software, all relevant methods are "by definition" in the *unshared* parts of the code. Methods that are *common* to both variants, such as elements #5 and #6, can thus be safely ignored, as they do not contribute to the *automatic save* but rather a previously present *user-triggered save* functionality, relied upon by the *automatic save file* feature.

[1] http://freemind.sourceforge.net

Unshared code may also include elements not relevant to the studied feature, e.g., element #7. In fact, Freemind's *automatic save file* feature is implemented by four methods, while there are 353 unshared methods between the variants. Thus, elements present in a specific part of the code should not be considered relevant automatically. Instead, existing feature location techniques should use this information to *enhance* their heuristics.

3. DIFF-SET BASED HEURISTICS

Feature location techniques are commonly evaluated by their *precision* – the fraction of elements deemed relevant among those reported, and *recall* – the fraction of reported relevant elements among all those deemed relevant.

In this section, we define a notion of a *diff set* – an artifact capturing *unshared* parts of the program of interest. We then describe two heuristics aimed to improve precision and recall of feature location techniques using *diff sets*: *filtering* and *score modification*.

DEFINITION 1. *Let P and $\bar{P}$ be program variants. A* diff set *of P compared to $\bar{P}$ (denoted by $\Delta P_{[\bar{P}]}$) is a set of all elements of P that do not have corresponding elements in $\bar{P}$. That is, $\Delta P_{[\bar{P}]}$ is a set of all elements of P that are either different or absent from $\bar{P}$.*

Filtering. Since we focus on *distinguishing* features of a program P, i.e., those that exist in P but not in another variant $\bar{P}$, we know a priori that all code of such features is present in P but absent in $\bar{P}$. That is, the implementation of such features fully resides in *diff sets*. Thus, all those elements retrieved by a feature location technique that do not reside in *diff sets* are false-positives: they do not contribute directly to the feature of interest and should not be returned to the user. As such, we propose to *filter* elements that do not reside in *diff sets* before returning feature location results to the user. *Filtering* can significantly improve the precision of existing feature location techniques without hurting their recall.

For the example in Section 2, the precision of a feature location technique that retrieves elements #1-8 as the result of locating *automatic save file* feature is 50%: only four shaded elements are really relevant to this feature (rather than to the *manual save file* feature which *automatic save file* uses). Considering the corresponding *diff set*, which contains five elements denoted by a solid-line boxes (elements #1-4, and 7), helps improve the precision to at least 80%: elements #5,6 and 8, even if retrieved during feature location can be *filtered* from the list of results.

In addition, *filtering* can help increase the number of relevant elements reported in the top ten highest ranked results – another often used metric inspired by the study in [20], which shows that users are generally unlikely to look at more than ten elements in a list. Such metric is suitable for the techniques that assign numeric

rank to the retrieved elements, representing their deemed relevance to the feature of interest. Leaving out elements that definitely do not contribute to the feature of interest can help push more relevant results up in the list.

Score Modification. While *filtering* is applicable to any feature location approach, focusing on a specific family of approaches (e.g., dynamic or static) can bring up additional improvements to the heuristics that they use.

We consider a family of feature location algorithms that employ an iterative, multi-staged program exploration approach, returning to the user a ranked list of relevant program elements. Such algorithms, e.g., *Dora* [7] or *Suade* [14], usually traverse the program structure, following program relationships such as method calls, data access/accessed-by relationships, type hierarchies and others. Analyzed program elements are scored based on their lexical and/or syntactical properties, and elements that are scored above a preset *relevance threshold* are returned to the user. Scoring is also used to determine which elements should be further traversed in the next iteration of the algorithm since they have a high chance to lead the analysis to additional relevant elements, and which are "dead ends" where the exploration should stop. For example, while traversing the call graph in Figure 1, a feature location technique might score each element based on its lexical similarity to a given query that describes a feature (e.g., *automatic save file*) and based on its structural proximity to other elements already determined to be relevant. The resulting score for each element represents the degree of relevance to the feature that is being detected.

Elements that reside in *diff sets* are more likely to be relevant to the feature of interest. Thus, we propose a *score modification* strategy aimed to increase the score of those elements in order to improve the recall of a given feature location technique. We increase the score of the elements in *diff sets proportionally to their original score*, calculated by the original feature location algorithm, instead of uniformly assigning a high score to all *diff set* elements. This is done in order to avoid a high number of false-positives, and thus a decrease in precision, as many elements in *diff sets* might not be relevant to the feature being located.

For the example in Figure 1, consider evaluating relevance of the element #3 w.r.t. the *automatic save file* feature, if the relevance threshold is 0.6. While the lexical similarity of this element to the "*automatic save file*" query, together with its structural proximity to other elements deemed relevant, is relatively low, e.g., 0.5, increasing the score of this element by 30% would push it above the relevance threshold. Thus, the feature location algorithm would determine it to be "relevant" to the feature in question, as desired. Increasing the score of the element #7 that has a lower initial score assigned by the original algorithm, e.g., 0.4, by 30% will not push it above the relevance threshold even though this element is also located in the *diff set*.

4. INITIAL EVALUATION

Subjects and Methodology. For evaluating our *diff-set* based approach, we are exploring a set of open source programs whose features were identified and studied earlier [13, 15]. These studies analyzed new functionality introduced by a specific software release (and thus absent in a previous version). The new functionality came in the form of *program patches* or *bug fixes* described in the projects' documentation or in online tracking systems.

Even though the main focus of our work is on features in different variants of a product family realized via code cloning, releases of a program mimic the qualities of such families when locating *distinguishing* features. Figure 3 sketches the cloning process in a family of related products, in which variants are created by duplicating a specific version and continuing its development independently from the original. For example, products P_2 and P_3 are created by cloning the existing product P_1 at points 2 and 4, respectively. After cloning, both new and existing products continue to involve independently from each other. Furthermore, product P_2 itself is cloned at point 5 to create another variant – P_4.

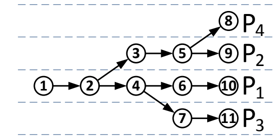

Figure 3: Cloned product variants.

When studying distinguishing features of a variant, instead of comparing it to another *variant* that lack those features, we can focus on comparing it to an earlier *version* that does not contain the features. For example, when studying distinguishing features of P_4, instead of comparing it to P_2 at point 9, we can compare it to P_2 at point 5, from which P_4 was cloned. In fact, comparing P_4 to P_2 in these two points yields the exact same *diff set*: the evolution of P_2 is irrelevant because we are only interested in those features that are present in P_4 and absent in P_2. Thus, for our analysis, we focus on versions of a program and use an earlier release to represent a variant that does not contain the feature of interest.

That is, we produced *diff sets* for the analyzed case studies by comparing the analyzed program variant P to an earlier release $\bar{P}$ which does not contain the feature of interest. To automate the comparison, we adapted a Java difference detection tool CHANGEDISTILLER [5] that extracts fine-grained source code changes between subsequent revisions of Java classes, based on calculating differences of their abstract syntax trees. As the result, we obtained a set of elements that are *new* or *modified* in P, compared to $\bar{P}$.

Observations. We analyzed the examples introduced by the study of Revelle and Poshyvanyk [13]. In their work, the authors focused on ten IR-based feature location approaches combined with static and dynamic techniques, and compared the results returned by each of the approaches to those produced manually by human evaluators. Specifically, the authors measured the percentage of elements perceived as "relevant" by human evaluators out of the top ten results reported by each of the evaluated techniques. The study demonstrated that contemporary feature location techniques are able to find only 12.5% to 30% of relevant elements in their top ten results, returning a large number of false-positives.

Unfortunately, the source code of the studied feature location approaches was not available to us. Thus, we were unable to measure the exact improvement in the precision and recall, as well as in the number of relevant elements returned by each of the techniques in its ten top ranked results, once the approaches have been enhanced with heuristics based on *diff sets*. However, we observed that around 80% of elements perceived as "not relevant" by the human evaluators in [13] corresponded to elements that did not change between program versions. This confirmed that our *diff set*-based heuristics can help improve the accuracy of a variety of feature location approaches.

We are now implementing the *diff-set*-based score modification for two tools whose source code was made available to us by their authors – *Dora* [7] and *Suade* [14]. An initial comparison of original vs. enhanced versions of these tools yields promising results.

5. DISCUSSION AND RELATED WORK

Numerous existing feature location approaches are extensively surveyed in [3, 16]. The approaches are largely divided into *dynamic* which collect information about a program at runtime, and *static* which do not involve program execution. Most dynamic approaches are based on analyzing two sets of traces – those produced by scenarios that activate a feature of interest and those that do not. Elements are considered "relevant" if they appear only in the former set. While execution trace partitioning is similar to our

partitioning of a program into *common* and *unshared* parts, the two strategies are orthogonal, potentially extending each other. Also, our approach does not require program execution.

Static feature location techniques look for desired results by leveraging program dependencies such as data or control flow or lexical similarity of the code to a query describing the feature of interest. These works do not explicitly address the issue of finding distinguishing features, and thus are orthogonal to our approach and can be extended by it.

Project documentation together with configuration management systems might help determine code that corresponds to features. However, detailed logs associating code with high-level features are rarely available. Several works attempted to discover features by analyzing *commit* operations. For example, *CVSSearch* [18] analyzed CVS log comments that describe the change made to the committed lines of code. Given a user query, the tool returns all lines of code that are mapped to comments containing at least one of the query words. Unlike *CVSSearch*, our approach does not rely on the availability of meaningful comments and is applicable even when no change tracking is present.

Yoshimura et al. [19] detect variability in a software product line using its evolution history. The work assumes that each product consists of individual components, and evolution entails an update to these. Components that are frequently modified together when creating one product from another are deemed to represent a product line variability point, which can also be perceived as a feature. We also attempt to detect elements that are introduced together and thus correspond to features that distinguish one product from another, as opposed to elements that are frequently committed together by the developer. However, we do not employ statistical techniques that require analyzing extensive historical data but rather suggest a simple heuristic for using *common* and *unshared* parts of code obtained from comparing program variants.

Kästner et al. [9] also deals with migration of a legacy system not designed as a software product line into a product line representation. Even though we share a similar goal of identifying product line features, our approach is for systems realized via code cloning, whereas the approach in [9] assumes "single-copy" legacy code.

6. CONCLUSIONS AND FUTURE WORK

Feature location is one of the most common activities undertaken by developers during software maintenance and evolution [12]. The goal of feature location is to retrieve those (and only those) elements that are relevant to the feature being detected. However, in practice, all feature location techniques are heuristic and thus unable to provide a clear cut distinction between relevant and irrelevant elements.

In this paper, we focused on locating *distinguishing* features of software product families realized via code cloning. That is, we aimed to find those features that are present in one variant of the program and absent in another. While in product families the mapping of features to product variants is usually well documented, locating the code of a feature in a given product variant is still challenging and, in many cases, inaccurate. Our approach is based on explicitly capturing the information obtained when comparing a product variant that has the feature of interest to another one that does not, and using that information to enhance the set of heuristics employed by existing feature location techniques. We believe that our approach helps increase the number of relevant elements identified by feature location, as well as to reduce the number of false-positive results.

As a future work, we intend to fully implement the outlined approach and evaluate its effectiveness on a set of realistic case studies. We are also interested in exploring additional usages of the collected information on program differences, as well as additional heuristics that can be derived for handling feature location in product lines.

7. REFERENCES

[1] D. Beuche. Transforming Legacy Systems into Software Product Lines. In *Proc. of SPLC'11 Tutorial*, 2011.

[2] P. C. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering. Addison-Wesley, 2001.

[3] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature Location in Source Code: A Taxonomy and Survey. *J. of Soft. Maintenance and Evolution*, 23(8), 2011.

[4] N. A. Ernst, S. M. Easterbrook, and J. Mylopoulos. Code Forking in Open-Source Software: a Requirements Perspective. *CoRR*, abs/1004.2889, 2010.

[5] B. Fluri, M. Wuersch, M. Pinzger, and H. Gall. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE TSE*, 33:725–743, 2007.

[6] P. Frenzel, R. Koschke, A. P. J. Breu, and K. Angstmann. Extending the Reflexion Method for Consolidating Software Variants into Product Lines. In *Proc. of WCRE'07*, pages 160–169, 2007.

[7] E. Hill, L. Pollock, and K. Vijay-Shanker. Exploring the Neighborhood with Dora to Expedite Software Maintenance. In *Proc. of ASE'07*, pages 14–23, 2007.

[8] C. Kästner and S. Apel. Integrating Compositional and Annotative Approaches for Product Line Engineering. In *Proc. of GPCE Wrksp. on Modul., Comp. and Gen. Tech. for PLE (McGPLE)*, pages 35–40, 2008.

[9] C. Kästner, A. Dreiling, and K. Ostermann. Variability Mining with LEADT. Technical report, Philipps University Marburg, 2011.

[10] C. W. Krueger. Easing the Transition to Software Mass Customization. In *Proc. of 4th Wrksp. on Soft. Product-Family Eng. (PFE)*, pages 282–293, 2002.

[11] T. Mende, R. Koschke, and F. Beckwermert. An Evaluation of Code Similarity Identification for the Grow-and-Prune Model. *Journal of Software Maintenance and Evolution: Research and Practice*, 21(2):143–169, 2009.

[12] V. Rajlich and P. Gosavi. Incremental Change in Object-Oriented Programming. *IEEE Software*, 21:62–69, 2004.

[13] M. Revelle and D. Poshyvanyk. An Exploratory Study on Assessing Feature Location Techniques. In *Proc. of IWPC'09*, pages 218–222, 2009.

[14] M. P. Robillard. Automatic Generation of Suggestions for Program Investigation. In *Proc. of ESEC/FSE-13*, pages 11–20, 2005.

[15] M. P. Robillard, D. Shepherd, E. Hill, K. Vijay-Shanker, and L. Pollock. An Empirical Study of the Concept Assignment Problem. Technical report, McGill University, 2007.

[16] J. Rubin and M. Chechik. A Survey of Feature Location Techniques. In I. Reinhartz-Berger et al., editor, *Domain Engineering: Product Lines, Conceptual Models, and Languages.* Springer, To appear.

[17] J. Rubin, A. Kirshin, G. Botterweck, and M. Chechik. Managing Forked Product Variants. In *Proc. of SPLC'12*, 2012.

[18] A. Y. Yao. CVSSearch: Searching through Source Code using CVS Comments. In *Proc. of ICSM'01*, pages 364–373, 2001.

[19] K. Yoshimura, F. Narisawa, K. Hashimoto, and T. Kikuno. FAVE: Factor Analysis Based Approach for Detecting Product Line Variability from Change History. In *Proc. of MSR'08*, pages 11–18, 2008.

[20] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller. Mining Version Histories to Guide Software Changes. In *Proc. of ICSE'04*, pages 563–572, 2004.