Clone Detection in a Product Line Context *

Thilo Mende, Felix Beckwermert
University of Bremen, Germany
{tmende,beckwermert}@informatik.uni-bremen.de

Abstract: Software Product Lines (SPL) can be used to create and maintain different variants of software-intensive systems by explicitly managing variability. Often, SPLs are organized as an SPL core, common to all products, upon which product-specific components are built. Following the so called grow-and-prune model, SPLs may be evolved by copy&paste at large scale. New products are created from existing ones and existing products are enhanced with functionalities specific to other products by copying and pasting code between product-specific code. To regain control of this unmanaged growth, such code may be pruned, that is, identified and refactored into core components upon success.

Clone detection offers effective means to identify duplicated source code. However, variablity in product lines, especially when targeting embedded devices, is often implemented using a preprocessor. This limits the applicable clone detection techniques to ones with lower precision. We describe how information about function locations can be used to improve the results of these token-based clone detectors.

1 Introduction

A software product line (SPL) is a set of software-intensive systems sharing a common, managed set of features. They satisfy the specific needs of a particular market segment or mission and are developed from a common set of core assets in a prescribed way [CN01]. According to Verhoef et al. [FV03], SPLs are either created *proactively*, i.e. designed upfront, or *reactively*, i.e. emerging from existing products, that possibly have been created using large-scale copy&paste programming.

The evolution of proactive SPLs is difficult to predict, thus an oscillation between generic and specific versions can be observed, also known as *configuration oscillation* [FV03]. Verhoef et al. propose to use a grow-and-prune approach to manage the consequences of the configuration oscillation. By explicitly allowing the uncontrolled growth, a customer's new needs are satisfied. After some time, a pruning phase identifies commonalities and a generic solution can be created. To reactively create an SPL from existing variants, one also has to identify parts that can be moved into the SPL core.

The identification of commonalities is thus required for both scenarios, and clone detection offers effective means to identify duplicated source code. However, SPLs, especially

*This work was performed as part of the project ArQuE (Architecture-Centric Quality Engineering), which is partially funded by the German Ministry of Education and Research (BMBF) under grant number 01 IS F14.

those written in C or C++ targeting embedded devices, often make extensive use of a preprocessor to implement variability, which limits the applicable techniques.

Overview. This paper describes how clone detection can be used to identify common source code that can be moved to the SPL core. The challenges applying clone detection in the presence of a preprocessor are described in Section 2, and Section 3 outlines an approach to address them using information about function location. Related work is described in Section 4, and Section 5 concludes.

2 Clone detection in product lines

Clone detection tools are used to search for duplicated source code. In the following, we consider tools that perform this search based on textual similarity, e.g. code that has been created using copy&paste. Unfortunately, these tools produce false positives: On the one hand, limitations in the programming language lead to code that is textually similar, but not refactorable. On the other hand, the developer may regard cloned code as not prunable, e.g. due to coding conventions. A high precision, i.e. a low false positive rate, is crucial for a practical adoption of the grow-and-prune approach, since the developer otherwise wastes time reviewing code he is not interested in.

Bellon et al.[BKA+07] and Bailey et al.[BB02] compared different clone detection tools. The best precision in both studies was achieved by syntax-based techniques. These identify code clones by comparing subtrees in abstract syntax trees, thus have to preprocess and parse the source code. This is a severe problem when the preprocessor is used to implement variability: Product-specific features are then enabled using conditional compilation, so after preprocessing only the code for one specific product can be parsed and analyzed.

In contrast, token-based techniques search for duplicated code on the non-preprocessed token stream. However, these tools have a much lower precision [BKA+07, BB02]. This aligns with our practical experiences, where we applied a token-based clone detector to an industrial SPL. The amount of false-positives as perceived by the developers was too high. Many of the detected clones started in one function and ended in another one, or were outside function definitions, and thus regarded as non-refactorable.

It became apparent that the developers were mostly interested in function-level clones. These are easier to refactor than arbitrary clones and the results are better to grasp using function names than source locations. We hence propose to filter the results of token-based clone detection using detailed information about function locations. Many of the false-positives mentioned above are then not existing.

As an additional aid for the developer, we then calculate the textual similarity of all functions sharing code. This value can be used to roughly estimate how much effort is necessary to merge the corresponding functions.

3 Improving token-based clone detection using function locations

As mentioned in the previous section, we only want to consider clones inside functions. Thus we have to map the clone candidates as determined by the detection tool onto the locations of function within the code. After that we calculate the textual similarity of each pair of functions sharing code. Finally, we build a graph containing the similarity information and filter the results to finally present them to the developer.

Function Locations To determine the similarity between functions, one obviously needs information about their location in the source code. Whenever a preprocessor is used, it is difficult to get this information reliably, because conditional compilation may change the location of a function depending on compile-time switches.

A heuristic approach to acquire this information non-preprocessed source code revealed severe problems in reliability. Instead, we use a C parser to extract this information from the preprocessed source code. The original, non-preprocessed source locations are available in the parser and used in the following steps.

Candidate Locations The candidate locations are used to determine which functions share at least some code. We use a token-based clone-detector (see section 2) similar to Baker's [Bak95] to detect clones in the original, non-preprocessed token-stream, with token-granularity. For each resulting clone pair, we determine the list of functions in each source code range and create a candidate pair for each function pair.

Similarity Calculation The edit distance described by Levenshtein [Lev66] measures the minimal amount of changes necessary to transform one sequence of tokens into a second sequence of tokens. For each candidate pair, as determined in the previous step, the Levenshtein distance is calculated.

Each function is represented as a normalized sequence. The normalization removes comments, line breaks and insignificant white spaces. The resulting edit distance then describes the number of tokens that have to be changed, removed or inserted to turn one function into the other. This can again be normalized to a relative value using the length of the corresponding sequence. We currently use the maximum length to get a relative value. While the Levenshtein distance is symmetric, the similarity is directed, since it describes the relative amount of tokens in one normalized sequence that is subsumed by the other.

Similarity Graph The next step is to build the similarity graph, a weighted graph containing all functions as nodes and all similarities as edges attributed with the similarity between them. Usually, the calculated similarities will be filtered to consider only similarities that might justify a refactoring. Besides a threshold for the similarity itself, a minimum number of equal tokens, a minimal length of each function, or other function-level metrics can be used to filter edges.

Aggregation and User Presentation To practically install a grow-and-prune approach, it is not feasible to periodically review all the similarities between functions. Instead, summaries are necessary to detect if and where it is advisable to start a restructuring effort. Therefore, similarity edges between functions are aggregated to a higher level. The hierarchy used to aggregate similarities can either be the structural decomposition on the

file system or the logical composition in the software architecture. We currently use the amount of subsumed tokens and the amount of similar functions as aggregated metrics.

The visualization of the potentially large result set is crucial for the practical adoption of the tool-based grow-and-prune approach. On the one hand, the developer is interested in the detailed similarities found by the analysis in order to merge similar functions. On the other hand, to decide whether a pruning phase is necessary and to monitor the evolution of an SPL, a higher-level view on the whole similarity graph is necessary.

A hierarchical adjacency matrix is used to display both aggregated similarities between hierarchical elements and similarities between functions. The developer is then able to explore the results by navigating from higher-level elements, such as directories or files, to the detailed list of similarities for one function.

4 Related Work

Using clone detection to manage and create product lines is mentioned in several papers, such as Baxter et al.[BC02]. In Kolb et al. [KMPY05], text-based clone detection is used to detect similarities in variants that are supposed to be merged into an SPL. The false-positive rate was too high so that the threshold for the minimal clone length was increased. However, we showed that this strategy might ignore interesting similarities at the function level[MBKM08].

There are three approaches similar to ours [MLM96, KST99, YMKI05]. The first one detects function clones by comparing a set of metrics for each combination of functions and then categorizes the results on an ordinal scale. This leads to an explosion of comparisons necessary to compare different products, and the ordinal scale makes an aggregation difficult. The second approach also uses metrics to identify similar lines in so called parameter files that are used to control variance of a telecommunication system. The third approach compares files with `diff` and uses the amount of shared lines as the similarity between files. Clone detection is used to reduce the necessary comparisons and the file granularity makes this approach scalable for large systems. However, measuring the similarity using lines might underestimate the similarity when only small changes to each line are made.

5 Conclusion

This paper described our experiences applying clone detection to support the creation and evolution of software product lines. An approach to enhance token-based clone detectors with information about function locations leads to promising results. It has been successfully applied to assess the restructuring potential of an industrial SPL with approximately 200 KLOC in more than 6.000 functions. A detailed description of this assessment, along with two other case studies, can be found in [MBKM08].

Further evaluations of different similarity measures, the aggregated metrics and the precision are left for future work.

References

[Bak95] B.S. Baker. On Finding Duplication and Near-Duplication in Large Software Systems. In *Proc. of 2nd WCRE*, pages 86–95, 1995.

[BB02] J. Bailey and E. Burd. Evaluating clone detection tools for use during preventative maintenance. In *Proc. of SCAM*, pages 36–43, 2002.

[BC02] I.D. Baxter and D. Churchett. Using Clone Detection to Manage a Product Line. In *ICSR7 Workshop*, 2002. PositionPaper.

[BKA+07] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE TSE*, pages 577–591, September 2007.

[CN01] P. Clements and L.M. Northrop. *Software Product Lines : Practices and Patterns*. Professional. Addison-Wesley, 2001.

[FV03] D. Faust and C. Verhoef. Software product line migration and deployment. *Journal of Software Practice and Experiences*, 33(10):933–955, August 2003.

[KMPY05] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi. A Case Study in Refactoring a Legacy Component for Reuse in a Product Line. In *Proc. of 21st. ICSM*, pages 369–378, 2005.

[KST99] A. Karhinen, M. Sandrini, and J. Tuominen. An approach to manage variance in legacy systems. In *Proc. of the 3rd CSMR*, pages 190–193, 3-5 March 1999.

[Lev66] V.I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. Technical Report 8, Soviet Physics Doklady, 1966.

[MBKM08] T. Mende, F. Beckwermert, R. Koschke, and G. Meier. Supporting the Grow-and-Prune Model in Software Product Lines Evolution Using Clone Detection. In *Proc. of the 12th CSMR*. IEEE Press, 2008. Accepted for publication.

[MLM96] J. Mayrand, C. Leblanc, and E.M. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proc. of 12th ICSM*, pages 244–253, 1996.

[YMKI05] T. Yamamoto, M. Matsushita, T. Kamiya, and K. Inoue. Measuring Similarity of Large Software Systems Based on Source Code Correspondence. In *PROFES*, pages 530–544, 2005.