

# Supporting the Grow-and-Prune Model in Software Product Lines Evolution Using Clone Detection\*

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

Gerald Meier  
Testo AG, Lenzkirch, Germany  
gmeier@testo.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.*

*This paper describes tool support for the grow-and-prune model in the evolution of software product lines by identifying similar functions which can be moved to the core. These functions are identified in two steps. First, token-based clone detection is used to detect pairs of functions sharing code. Second, Levenshtein distance measures the textual similarity among these functions. Sufficient similarity at function level is then lifted to the architectural level.*

*The approach is evaluated by three case studies, one using an open source email client to simulate the initial creation of an SPL, and two monitoring existing industrial product lines from the embedded domain.*

**Keywords:** *Software maintenance, Software reusability*

## 1 Introduction

*Software Mitosis*, as described in [11] is the uncontrolled natural growth of software systems by copying

---

\*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.

and pasting components or whole systems at the large. Among the reasons for the growth are different customers with different requirements, local modifications in global organizations, and variations in the underlying hardware for embedded systems. Over time, it becomes more and more difficult to manage the variety of systems.

A software product line (SPL) is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [7]. There are different approaches for SPLs to organize reuse for families of systems. Various implementation techniques such as conditional compilation, parameterization, generics, and code generation have been proposed to implement variability in SPLs in a maintainable way.

However, since the evolution of an SPL is difficult to predict, an oscillation between generic and specific versions can be observed, also known as *configuration oscillation* [11]. Verhoef et al. propose to use a grow-and-prune approach to manage the consequences of software mitosis and 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.

In the pruning phase, similar code in different variants has to be identified in order to be reused as a core asset. The identification should be automated as far as possible to be able to determine when to start such a pruning phase. One of the metrics proposed by Verhoef et al. to identify such reusable components is the *mitosis delta*, i.e., the difference of two units as a percentage. They use clone detection to calculate this mitosis delta, but without providing technical details.

**Contributions.** This paper proposes to use the Levenshtein distance to measure similarity between functions and thereby calculating the mitosis delta.

Clone detection is used to keep the computational costs tractable for large systems. Various metrics that aggregate the similarity at the architectural level are proposed that allow to determine the need for a pruning phase.

Three case studies are then used to show how the grow-and-prune approach can be used in practice both for the initial creation of an SPL and the monitoring of existing SPLs.

**Overview.** The remainder of this paper is organized as follows. First, our approach is described in Section 2, and the implementation is outlined in Section 3. The approach is evaluated for three example systems in Section 4. Then, related work is discussed in Section 5. Finally, Section 6 concludes.

## 2 Approach

To practically install a grow-and-prune approach, effective tool support is necessary to identify reusable components. Verhoef et al. propose to use the mitosis delta between units, but without specifying how it is measured exactly and what granularity of units is used.

In this paper, similarity is measured between functions, i.e., first the goal is to identify all functions with a significant similarity. The mitosis delta is then the part of two functions that are not similar.

Using the function level has several advantages:

- Stack size is a critical issue for the embedded domain, so using the *extract method* refactoring is often not an option.
- A considerable amount of copied code at the file level, i.e., outside functions, can not be avoided, for instance, includes and declarations.
- In our experience, the communication with developers is much easier using function names instead of code locations because they form logical abstractions.
- If similarities are tracked over several versions and variants of a product line, the names of the functions may provide additional evidence for correspondences between code segments among versions and variants.
- Additional metrics about functions, e.g., commonalities and differences at call sites can be taken into account when results are evaluated.

To compute the similarity between all possible combinations of functions in all products, ignoring functions within the same product, one has to compare

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^n F_i \cdot F_j$$

functions, where  $n$  is the number of products and  $F_i$  is the number of functions in product  $i$ . Under the assumption that all products have a similar amount of functions, this evaluates to

$$\frac{n(n-1)}{2} \cdot F_i^2$$

Thus, calculating all possible similarities is not feasible for real systems. The following sections describe in detail how the computational costs can be reduced using clone detection.

### 2.1 Function Locations

For each source file, the following line and column information about functions is useful:

- beginning of the signature
- beginning of the declaration
- beginning of the body
- end of the body

To calculate the similarity between function bodies, only the latter two are necessary, however the first one is useful to be able to compare function signatures (see Section 4.3), and the second is necessary to get the name of a function.

Whenever a preprocessor is used, it is generally impossible to generate this information reliably, because conditional compilation may change the location of a function depending on compile-time switches.

One possibility, that was implemented in an earlier version of our tool set, is to use heuristics to acquire this information from the non-preprocessed source code. However, tests revealed severe problems in reliability using this approach.

The current implementation uses 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. Whenever compile-time switches change this information, the larger range is taken into account. However, by using the preprocessed source code, this implementation considers only function locations that are not disabled using compile time switches, thus it is necessary to pay attention to the actual preprocessor configuration.

### 2.2 Candidate Locations

The candidate locations are used to determine for which functions the similarity has to be calculated,

and therefore one has to balance between precision and speed. On the one hand, the goal is to reduce the number of candidates so that unnecessary similarity calculations are avoided. On the other hand, this phase has to be significantly faster than calculating all possible similarities.

We use a token-based clone-detector similar to Baker’s [2] to detect clones in the original, non-preprocessed token-stream, with token-granularity, as described in [17, 10]. 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.

Our token-based clone-detector has a lower precision than syntax-based techniques [1, 17, 10, 6] However, using it for SPLs is appealing for several reasons. First, it works linear in time and space, thus it is scalable for large (families of) systems. Second, it works on the original, non-preprocessed source-code. This is especially important for SPLs, since variability is often implemented using conditional compilation. Third, it is used only to get a first hint, thus recall is more important than precision. Its results will be used to compute Levenshtein distance for whole functions in which the identified copied code occurs, and hence syntactic borders are taken into account and most false positives will be filtered out.

We note that the overall approach is independent of the token-based clone-detection. Any technique that generates a list of function candidates can be used, e.g., clone detectors based on syntax or metrics. For smaller systems, it may also be possible to calculate all possible combinations, e.g., when most of the functions are contained in a framework and the variants have only few functions.

### 2.3 Similarity Calculation

The edit distance described by Levenshtein [18] 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 (LD) is calculated.

Each function is represented as a normalized sequence  $sf_x = \text{norm}(f_x)$ . The normalization removes comments, line breaks and insignificant white spaces. The resulting edit distance  $\Delta f_{x,y} = \text{LD}(sf_x, sf_y)$  then describes the number of tokens that have to be changed to turn function  $f_x$  into  $f_y$ .

This can be normalized to a relative value using the length of the corresponding sequence  $|sf_x| = \text{len}(sf_x)$ . We currently use the maximum length to get a relative

value:

$$\text{sim}(f_x, f_y) = \frac{\max(|sf_x|, |sf_y|) - \Delta f_{x,y}}{|sf_x|}$$

While the Levenshtein distance is symmetric, similarity *sim* is directed, since it describes the relative amount of tokens in sequence  $sf_x$  that is subsumed by  $sf_y$ . The amount of shared code, i.e., the amount of tokens of  $f_x$  that is contained in  $f_y$ , is

$$\text{shared}(f_x, f_y) = |sf_x| \cdot \text{sim}(f_x, f_y)$$

We currently compare the sequence of characters in each function to calculate the Levenshtein distance, which has the advantage that small modifications in identifiers result in a large relative similarity. However, computing Levenshtein distance has a time complexity of  $O(|sf_x| \cdot |sf_y|)$ , thus the length of the strings compared has a large impact on the time. An alternative, although currently not implemented, is to compare the sequence of token values instead of characters. In the following, all similarities are measured on the character level.

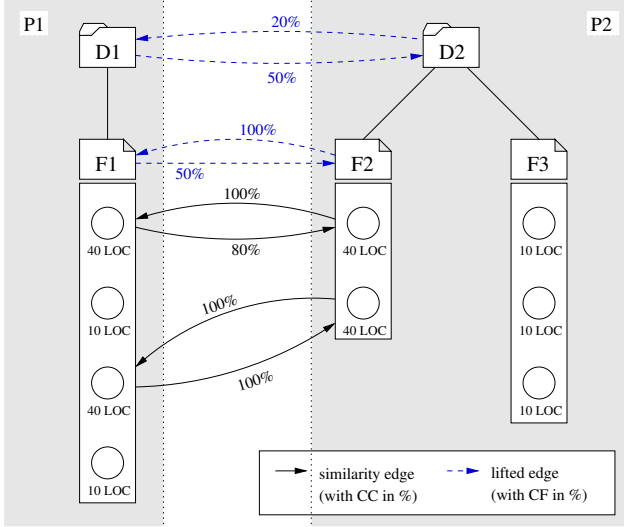
### 2.4 Similarity Graph

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 of equal tokens or a minimal length of each function can be used to filter edges. Possible refactorings are domain and application depended, thus the filters used in our case studies are described in Section 4. In the following, two functions that are connected by a similarity in the similarity graph – i.e., after the filtering – are considered *cloned functions*.

Similarity edges are directed and attributed with  $\text{sim}(f_x, f_y)$  and  $\text{shared}(f_x, f_y)$ . To abstract from the kinds of tokens used to calculate Levenshtein distance LD,  $\text{sim}(f_x, f_y)$  can be used to estimate the amount of lines from  $f_x$  subsumed by  $f_y$ . Although this value is not accurate, it may be easier to grasp for the developer.

### 2.5 Aggregation

To practically install a grow-and-prune approach, it is not feasible to review each function’s similarities regularly. Instead, summaries are necessary to detect if and where it is advisable to start a restructuring effort.



**Figure 1. Similarity Graph with (some) aggregated edges**

Therefore, similarity edges between functions have to be 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.

The following questions should be answerable using the aggregated metrics, where  $A$  and  $B$  are composite units such as logical modules or physical files or directories:

- How much code can potentially be pruned when  $A$  is merged into  $B$ ?
- How much functionality of  $A$  is subsumed by  $B$ ?

To answer these questions, the following aggregated metrics are defined:

**CommonCode (CC)** The absolute and relative amount of tokens in  $A$  subsumed by  $B$ . If a function in  $A$  has more than one cloned function in  $B$ , only the largest amount of tokens is considered, so that the relative amount of shared tokens cannot be larger than 100%.

**CommonFunctions (CF)** The question whether one piece of code subsumes the functionality of another piece of code is, of course, undecidable. Yet, similar functions offer similar functionality at least at the abstract level (their differences and different scope may lead to different behavior, yet the behavior of the shared code skeleton is similar). As a rough measure, we use the absolute

	D1	F1	f1	f3	D2	F2	f5	f6
▷ D1		-	-	-	2 50%	2 50%	35 80%	40 100%
▷ F1	-		-	-	2 50%	2 50%	35 80%	40 100%
▷ f1	-	-		-	35 80%	35 80%	35 80%	-
▷ f3	-	-	-		40 100%	40 100%	-	40 100%
▷ D2	2 20%	2 100%	35 100%	40 100%		-	-	-
▷ F2	2 20%	2 100%	35 100%	40 100%	-		-	-
▷ f5	35 100%	35 100%	35 100%	-	-	-		-
▷ f6	40 100%	40 100%	-	40 100%	-	-	-	

**Figure 2. Hierarchical adjacency matrix for Figure 1 showing absolute and relative degrees of similarity; entries for functions show CommonCode (CC), entries for files and directories show CommonFunctions (CF).**

and relative number of functions in  $A$  that are subsumed by  $B$ , i.e., that have a similarity edge in the similarity graph. To avoid values above 100%, multiple edges from functions in  $A$  to  $B$  are ignored. This metric intentionally ignores the size of the similar functions as it's goal is to express how much functionality is subsumed, not how much code.

An example for an aggregated similarity graph can be found in Figure 1. Edges between different levels in the hierarchy, i.e., between files and directories, are not shown to keep the figure concise. Nevertheless, the aggregation is calculated between all hierarchy levels.

The example displays the length of functions and CC in lines, which is only an estimation when the similarity is calculated using characters or tokens.

## 2.6 Visualization

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. A detailed view is supported using a list of functions, each associated with its cloned functions. Additional information about functions, e.g., their size and how often they are used, is helpful as well as the ability to show the differences between them.

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 provides such a higher-level view. An example for the aggregated similarity graph in Figure 1 can be found in Figure 2. It uses the metric CommonFunctions (CF) for aggregated edges and has to be read as follows: the entry in row  $n$  is subsumed by the entry in column  $m$ . For instance,  $F2$  is completely subsumed by  $F1$ , because all its functions are contained in  $F2$ . By displaying both the absolute and relative values, it is possible to quickly identify components that are potential candidates for a refactoring.

As the metric  $CF$  is not interesting for aggregated edges between functions and files or directories,  $CC$  is used for these edges.

### 3 Implementation

Our implementation is based on the *Bauhaus* toolkit<sup>1</sup> for software analysis. The resulting similarity graph is visualized using *GraVis*, the *Bauhaus* graph visualizer. The similarity graph may be exported to the Graph Exchange Language<sup>2</sup> (GXL). With the help of *GraVis*, users can interactively browse the similarity graph, inspect the sources of a clone edge in a built-in text viewer with textual difference highlighting and analyze the results in context of other automatically extracted code dependencies, such as call relations and variable usage.

The *Bauhaus* clone detector we used in this paper, *clones*, implements a variation of Baker’s token-based algorithm [2]. The main difference to Baker’s original technique is that *clones* does not summarize all tokens in the same line and, hence, is independent of the source code layout. Also, *clones* has the ability to identify clones with inconsistent replacement of identifiers and literals (we have not used this feature in this paper, however). In addition to *clones*, *Bauhaus* offers three more clone detectors. One of which is *ccdimpl*, a variation of Baxter’s syntax-based approach [5]. A new clone detection technique based on linearized syntax trees is implemented by *clast* for parse trees and *cpdetector* for abstract syntax trees as described in [17, 10]. In future research, we may investigate pros and cons of replacing *clones* with the alternative *Bauhaus* clone detectors.

### 4 Case Study

The following sections evaluate the proposed approach using three different product families, one open-

<sup>1</sup><http://www.axivion.com/>

<sup>2</sup><http://www.gupro.de/GXL/>

Product	SLoc	Nr. of Functions
Sylpheed	71,736	2,741
Sylpheed2	80,353	2,893
Claws Mail	129,310	4,449
$\Sigma$	281,399	10,083
Number of combinations		33 Mio

**Figure 3. Sylpheed variants’ source lines inside functions and number of functions**

source and two industrial ones. As a matter of fact, the open-source system is not really an SPL. It is very difficult to find SPLs in the open-source domain. Yet, even though the open-source system only resembles an SPL, we added it as a case study so that our results can be reproduced by other researchers. It may also be valuable for future benchmarks.

#### 4.1 Sylpheed

The open-source email-client Sylpheed<sup>3</sup> was started in 2000 using the GUI toolkit GTK. In 2005, a fork, Sylpheed2, was created to use the new version of this toolkit, GTK2. Claws Mail<sup>4</sup> was originally a development version of Sylpheed, but was forked using a pre-release version of Sylpheed2. Claws Mail was first released using the GTK toolkit in May 2002, the first release based on GTK2 in March 2005.

We use these three variants to simulate the initial creation of an SPL from existing products evolved from the same code base, even though it is very unlikely that this will happen in practice. Nevertheless, using this software has the advantage that it is freely available, so other approaches can be compared with the one presented here. Furthermore, each variant makes intensive use of the preprocessor to implement portability between operating systems and to implement variable features such as IMAP, SSL, etc., which is common practice for real SPLs. Claws Mail alone uses 23 compile-time switches for variable features, resulting in over 100 different conditions for `#ifdefs`.

All three products together contain 400K SLOC, measured using `sloccount`<sup>5</sup>. The lines of code inside functions and the number of functions per product can be found in Figure 3.

The similarity graph is calculated using the following filters:

- The clone detector reports only clones with at least 30 tokens. 979,238 clone pairs are found.

<sup>3</sup><http://sylpheed.sraoss.jp> Version 1.0.6 and 2.4.7

<sup>4</sup><http://www.claws-mail.org> Version 3.0.2

<sup>5</sup><http://www.dwheeler.com/sloccount/>

	Sylpheed	Sylpheed2	Claws-mail
Sylpheed		1454 (53 %)	922 (33 %)
Sylpheed2	1456 (50 %)		838 (28 %)
Claws-mail	972 (21 %)	888 (19 %)	

**Figure 4. Similarity measured in functions between Sylpheed variants**

	Sylpheed2	Claws-mail
Sylpheed	1454 (53%)	922 (33%)
vcard.c	100%	53%
addrcache.c	87%	46%
md5.c	–	100%
editaddress.c	93%	68%
prefs_customheader.c	88%	41%
displayheader.c	100%	100%
mainwindow.c	72%	43%
main.c	27%	–
gtkscstree.c	–	68%
summary_search.c	–	–
base64.c	100%	100%
mbox.c	100%	16%
uuencode.c	100%	100%
...	...	...

**Figure 5. Assessing the potential to merge Sylpheed into Sylpheed2 or Claws Mail**

- These clones affect 29,439 function pairs for which the similarity is calculated.
- All function pairs with a similarity greater or equal to 70 % are considered as cloned functions. This results in 5,158 similarity pairs in 4,361 different functions.

The aggregated metric CommonFunctions (CF) for the three variants of Sylpheed can be found in Figure 4. Despite the different GUI toolkits and the fact that the variants have evolved for several years, the commonalities between them are remarkably high. For example, 1,454 functions, i.e., 53 %, in Sylpheed are cloned in Sylpheed2.

Now consider the following use case: the goal is to assess whether Sylpheed’s functionality can be merged into either Sylpheed2 or Claws Mail. This can be done using the CommonFunctions metric to compare the amount of functions in Sylpheed that are subsumed by the other products. An excerpt of the adjacency

min. Length	Nr. Clones	Candidates	Cloned
30	979,238	29,439	5,158
40	445,081	17,189	4,294
50	213,142	11,405	3,709
60	124,111	8,121	3,220
70	63,609	5,596	2,688
80	38,771	4,342	2,426
90	24,956	3,644	2,234
100	15,053	3,102	2,095

**Figure 6. Influence of the minimal clone length in token**

matrix displaying the relative metric can be found in Figure 5. Out of the 94 files that were analyzed, 11 files are completely subsumed by Sylpheed2 and 7 by Claws Mail. When all files with at least 80 % similar functions are considered, 25 or 10, respectively are cloned. This first assessment suggests that it is might be worthwhile to merge Sylpheed and Sylpheed2.

As described in Section 2.2 and Section 2.3, it is necessary to keep the number of comparisons low, since computing the similarity is expensive. It is therefore interesting to analyze the influence of the minimal token length used by the clone detector to detect candidate locations. In Figure 6, the amount of candidate functions and cloned functions, i.e., functions with a similarity higher than 70 %, depending on the minimal number of tokens necessary can be found. Although the number of comparisons increases significantly for shorter token lengths, setting the threshold too high results in false negatives.

## 4.2 Automotive Device

In this section, we summarize the results of an earlier industrial case study in which we evaluated an extension of the reflexion method to SPLs [12]. We report only those results relevant to the context of this paper.

In the earlier case study, we analyzed industrial software of a family of embedded systems from the automotive domain written in C. Company-sensitive details are omitted as much as possible. The software is embedded in a control device and deployed by different car manufacturers. The diversity of the hardware and feature set of this family of control devices is mirrored in the software. An older generation of this SPL that we analyzed is implemented by a mixture of preprocessor directives and branches in the version control system. The family of systems consists of hundreds of variants. To evaluate our proposed method, we selected two variants of this SPL, one variant for a low-end car,

variant	SLOC	func	k-func	f-func
H	106,383	1,538	584	954
L	64,091	1,017	520	497

**Table 1. System characteristics**

$L$ , and one for a high-end car,  $H$ , of two different car manufacturers.

Despite the differences of the supported features, the architectures are similar. The SPL architecture is organized as a set of *kernel modules* providing hardware abstraction intended to be largely reusable across products and a set of *feature modules*, which are specific to a product. Because of hardware differences, the kernel modules are not identical across variants. Table 1 lists some characteristics of these two variants. *SLOC* is the number of physical sources lines measured by `sloccount`. Column *func*. is the number of functions defined in the variants. The number of functions of kernel modules is listed in column *k-func* and those of feature modules in *f-func*.

We detected similar functions of  $L$  in  $H$ . Each function  $f$  is associated with a set of corresponding identical functions  $F_I = \{f' | \text{sim}(f, f') = 1\}$  and corresponding similar functions  $F_S = \{f' | \theta \leq \text{sim}(f, f') < 1\}$  of the other variant. The following situations may occur:

- I1  $|F_I| = 1$ : there is one identical correspondence for  $f$  in the other variant
- I2  $|F_I| > 1$ : there are multiple identical correspondences for  $f$  in the presence of function clones
- S1  $|F_I| = 0 \wedge |F_S| = 1$ : there is one varied correspondence for  $f$ ; the difference of  $f$  and this candidate is an indicator for a potential variability
- S2  $|F_I| = 0 \wedge |F_S| > 1$ : there are multiple varied candidates for  $f$ ,
- U otherwise: no correspondence can be found

We can gather the frequency distribution of the above classes for all functions of the variant to be analyzed. Because this distribution depends upon the threshold of similarity,  $\theta$ , we will experiment with different values for  $\theta$  in the range of 0.6...0.95.

We compared all kernel functions of  $H$  to all kernel functions of  $L$  and all feature functions of  $H$  to all feature functions of  $L$ . Tables 2 and 3 show the results of the respective separate analysis. As expected, the correspondence between kernel functions is much stronger than between feature functions. Yet, about a third of the kernel functions does not have any correspondence, which indicates a high degree of variance even in the kernel modules. The variance is due to adaptations to a different hardware platform.

$\theta$	I1	I2	S1	S2	U
0.6	168	3	121	41	222
0.7	168	3	98	31	255
0.8	168	3	69	21	294
0.9	168	3	41	10	333
0.95	168	3	27	0	357

**Table 2. Results for whole kernel modules**

$\theta$	I1	I2	S1	S2	U
0.6	11	8	23	15	521
0.7	11	8	23	5	531
0.8	11	8	19	0	540
0.9	11	8	13	0	546
0.95	11	8	8	0	551

**Table 3. Results for whole feature modules**

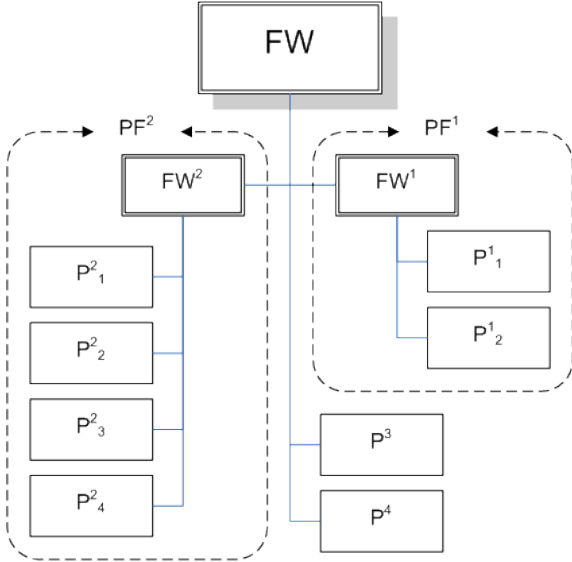
As Table 3 shows 19 (= 11+8) product-specific functions are identical between the two variants and may be moved to the kernel. Depending upon the similarity threshold  $\theta$ , there are potentially in between 23+15 and 8+0 additional functions that may be moved to the kernel after some unification. In particular, if  $\theta = 0.95$ , the results should be quite reliable. To see how reliable the results are for lower values of  $\theta$ , we validated manually the correspondence of functions classified as  $S1$  between the two variants as a whole. Of the 175 functions in  $S1$  for  $\theta = 0.7$ , 21 have only minor syntactic differences and could actually be classified as  $I1$  instead. For these cases, the textual Levenshtein distance does not provide enough abstraction. Another 121 functions in this class turn out to be real variants after manual validation, whereas 33 can be considered false positives. This manual inspection indicates a relatively high reliability even for varied functions.

It is also interesting to see that there are complete functions clones within  $L$  both in kernel as well as a function modules as indicated by values for  $I2$  greater than 0.

### 4.3 Testo AG

Testo is a leading vendor in the field of climate and gas analysis. Their products are developed as an SPL since 2001, as described in [15]. The basis of their SPL is a product-independent framework, with an additional framework layer for each product family. Before using the SPL, new products were created by copying&pasting similar products. Since then, the existing products were merged and new products are created by reusing as much framework functionality as possible.

The goal of the analysis presented here is to determine to which degree the merging phase has been completed successfully, to monitor the evolution of the



**Figure 7. Testo's Product Hierarchy**

Product	SLoc	Nr. of Functions
<i>FW</i>	29,488	861
<i>FW</i> <sup>1</sup>	10,327	325
<i>P</i> <sub>1</sub> <sup>1</sup>	17,392	504
<i>P</i> <sub>2</sub> <sup>1</sup>	45,659	1,133
<i>FW</i> <sup>2</sup>	2,187	84
<i>P</i> <sub>1</sub> <sup>2</sup>	3,067	119
<i>P</i> <sub>2</sub> <sup>2</sup>	1,422	71
<i>P</i> <sub>3</sub> <sup>2</sup>	16,249	703
<i>P</i> <sub>4</sub> <sup>2</sup>	2,645	121
<i>P</i> <sup>3</sup>	15,351	581
<i>P</i> <sup>4</sup>	55,737	1,856
∑	199,524	6358
Number of combinations		16,8 Mio

**Figure 8. Testo variants' source lines inside functions and number of functions**

product line to identify the need for a pruning phase, and to determine reusable components that are currently implemented in product-specific source code.

First tests with only token-based clone detection were not successful, because the false positive rate was far too high. On the other hand, syntax-based techniques have problems with the high degree of variability in framework components. Additionally, while discussing the results of the token-based clone detection with the developers, it became apparent that the usage of functions as the unit of comparison makes it much easier for developers to evaluate whether a candidate can in fact be refactored.

Eight products and three core libraries were part

of this analysis, totalling in 200,000 lines of code inside functions and 6,358 functions. The detailed distribution of code and functions can be found in Figure 8. The total amount of necessary comparisons among functions in all products is approximately 16.8 million. The products and their hierarchy, i.e., the separation into framework, product-families, and products, can be found in Figure 7.

The following rules are currently used to identify similarities that are indeed refactorable for Testo:

- The clone detection reports only clones with at least 60 tokens. This results in 99,664 clone pairs.
- These clone pairs affect 3,343 function pairs for which the similarity is calculated.
- When only pairs with a similarity higher than 70 % and a minimal function length of 100 token are considered, 826 cloned functions are found.

In the list representation of the similarity graph, the textual similarity of signatures is used to categorize similarities as described in [23] enabling the developer to determine how easy it is to unify the code.

The overall result, as seen in Figure 9, is that a pruning phase at the global level is not promising at the moment. The details for *PF*<sup>2</sup>'s framework and products are omitted due to space constraints. Relative to the amount of code, only a small fraction can be refactored, at least when stack size is taken into account. The somewhat stronger similarity between some products, e.g., between *P*<sup>3</sup> and *P*<sup>4</sup> were expected by Testo. These similarities are functions belonging to code from the compiler vendor that provides access to the underlying hardware. It will be refactored in the next iteration of the framework development.

Using the filtering mechanisms described above, only few false positives were found, so that the approach can and will be used regularly to monitor the evolution of Testo's product line in order to determine the necessity for a pruning phase.

## 5 Related Work

The grow-and-prune model, the basic approach of this paper, was first described in [11], where a case study describes its successful adoption for a global financial application. However, technical details, such as the calculation and aggregation of the mitosis delta are not provided. In [20], software mitosis and the configuration oscillation is also observed in the evolution of a product line, and approaches are described that use higher-level architectural solutions.

	<i>FW</i>	<i>PF</i> <sup>1</sup>	<i>FW</i> <sup>1</sup>	<i>P</i> <sub>1</sub> <sup>1</sup>	<i>P</i> <sub>2</sub> <sup>1</sup>	<i>PF</i> <sup>2</sup>	<i>P</i> <sup>3</sup>	<i>P</i> <sup>4</sup>
<i>FW</i>		5 (0%)	1 (0%)	3 (0%)	1 (0%)	11 (1%)	4 (0%)	8 (0%)
<i>PF</i> <sup>1</sup>	5 (0%)		11 (0%)	175 (8%)	173 (8%)	28 (1%)	69 (3%)	97 (4%)
▷ <i>FW</i> <sup>1</sup>	1 (0%)	11 (3%)		–	11 (3%)	15 (4%)	17 (5%)	29 (8%)
▷ <i>P</i> <sub>1</sub> <sup>1</sup>	3 (0%)	162 (32%)	–		162 (32%)	11 (2%)	24 (4%)	30 (5%)
▷ <i>P</i> <sub>2</sub> <sup>1</sup>	1 (0%)	186 (16%)	11 (0%)	175 (15%)		2 (0%)	28 (2%)	38 (3%)
<i>PF</i> <sup>2</sup>	11 (1%)	41 (3%)	16 (1%)	23 (2%)	5 (0%)		20 (1%)	18 (1%)
<i>P</i> <sup>3</sup>	4 (0%)	57 (9%)	17 (2%)	21 (3%)	26 (4%)	15 (2%)		175 (30%)
<i>P</i> <sup>4</sup>	9 (0%)	89 (4%)	27 (1%)	28 (1%)	39 (2%)	13 (0%)	185 (9%)	

**Figure 9. Similarity measured in cloned functions (absolute and relative) for Testo’s variants**

Using clone detection to manage and create product lines is mentioned in several papers, such as [4]. In [16], text-based clone detection is used to detect similarities in variants that are supposed to be merged into an SPL. However, the false-positive rate was too high so that a minimum clone length of 25 lines is used. This, however, might ignore interesting similarities at the function level, as shown in Section 4.1.

The migration towards a software product line is also supported using clone detection in [23]. They use a categorization based on the similarity of the signature. Finally, an early version of this approach is used in [12] to incrementally recover a product line architecture.

Clone detection can be distinguished based on the type of information the analysis is based on and in the algorithm used. The information used ranges from line- or token-based textual comparisons over comparison of abstract syntax trees like in [5] to comparison of program dependency graphs and metric based techniques [19]. Depending on the algorithm used, the complexity varies from linear in space and time to quadratic. According to a study by Bellon[6], the results of the diverse techniques differ in the types of clones they detect and the trade-off between precision and recall. Syntax-based techniques offer the best precision, i.e., the lowest false-positive rate, while token-based techniques have a higher recall at a (much) lower precision. Levenshtein distance has been used to identify clones in object oriented systems [3] and web pages [8], and to rate clones, e.g. in [9, 13].

A similar problem of identifying similar functions among software systems arises in evolutionary analy-

sis [21]. Here, the correspondence of functions among different versions must be established to track certain evolutionary characteristics of interest such as length, complexity and other metrics. This problem is also known as origin analysis [24].

Three approaches similar to ours are [19, 14, 22]. 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 using `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 between files.

## 6 Conclusion

This paper provides technical details how the grow-and-prune model can be supported using clone detection and detailed function locations. The Levenshtein distance, used to measure similarity between functions, provides an intuitive measurement about the similarity of two functions. Using functions as the lowest unit that is compared is a useful abstraction, since it is much easier to communicate results to the developer. Mea-

asuring similarity of the non-preprocessed source code meets the expectations of developers when reviewing results and is crucial for SPLs where conditional compilation is often used to create variability. However, without the usage of clone detection techniques, the computation would be infeasible, as shown in the case studies.

The precision of our approach, i.e. the amount of false positives, has not been quantitatively evaluated, although feedback from the users has been promising. An evaluation of the false positive rate, together with a comparison of different similarity measurements, is left for future work.

The approach suffers when functions are very large, since the potential to remove large blocks of code is currently ignored. An extension to address this problem, together with an extension to C++ and Java is also left for future work.

## References

- [1] J. Bailey and E. Burd. Evaluating clone detection tools for use during preventative maintenance. In *Workshop Source Code Analysis and Manipulation*, pages 36–43, 2002.
- [2] B. S. Baker. On Finding Duplication and Near-Duplication in Large Software Systems. In *Proc. of 2nd WCRE*, pages 86–95, 1995.
- [3] E. Balazinska, M. and Merlo, B. Dagenais, M. and Lague, and K. Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *Proc. of 7th WCRE*, pages 98–107, 2000.
- [4] I. D. Baxter and D. Churchett. Using clone detection to manage a product line. In *ICSR7 Workshop, Industrial Experience with Product Line Approaches*, 2002. PositionPaper.
- [5] I. D. Baxter, A. Yahin, L. Moura, M. Sant’Anna, and L. Bier. Clone Detection Using Abstract Syntax Trees. In *Proc. of 14th ICSM*, pages 368–378, 1998.
- [6] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, pages 577–591, Sept. 2007.
- [7] P. Clements and L. M. Northrop. *Software Product Lines : Practices and Patterns*. Professional. Addison-Wesley, 2001.
- [8] G. Di Lucca, M. Di Penta, and A. Fasolino. An approach to identify duplicated web pages. In *International Computer Software and Applications Conference*, pages 481–486, 2002.
- [9] E. Duala-Ekoko and M. Robillard. Tracking code clones in evolving software. In *Proc. of 29th ICSE*, pages 158–167, 2007.
- [10] R. Falke, P. Frenzel, and R. Koschke. Empirical evaluation of clone detection using syntax suffix trees. *Empirical Software Engineering Journal*, 2007. Submitted for publication.
- [11] D. Faust and C. Verhoef. Software product line migration and deployment. *Journal of Software Practice and Experiences*, 33(10):933–955, Aug. 2003.
- [12] P. Frenzel, R. Koschke, A. Breu, and K. Angstmann. Extending the reflection method for consolidating software variants into product lines. In *Proc. of 14th WCRE*, 2007.
- [13] C. J. Kapsner and M. W. Godfrey. Supporting the analysis of clones in software systems: a case study. *Journal of Software Maintenance and Evolution*, 18(2):61–82, 2006.
- [14] 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.
- [15] R. Kolb, I. John, J. Knodel, D. Muthig, U. Haury, and G. Meier. Experiences with product line development of embedded systems at Testo AG. In *Proc of the 10th SPLC*, pages 172–181, 2006.
- [16] 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.
- [17] R. Koschke, R. Falke, and P. Frenzel. Clone detection using abstract syntax suffix trees. In *Proc. of 13th WCRE*, pages 253–262, 2006.
- [18] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. Technical Report 8, Soviet Physics Doklady, 1966.
- [19] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proc. of 12th ICSM*, pages 244–253, 1996.
- [20] C. Riva and C. Del Rosso. Experiences with software product family evolution. In *Software Evolution, 2003. Proceedings. Sixth International Workshop on Principles of*, pages 161–169, 1-2 Sept. 2003.
- [21] Z. Xing and E. Stroulia. Understanding phases and styles of object-oriented systems’ evolution. In *Proc. of 20th. ICSM*, pages 242–251, 2004.
- [22] 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.
- [23] K. Yoshimura, D. Ganesan, and D. Muthig. Defining a strategy to introduce a software product line using existing embedded systems. In *EMSOFT ’06: Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 63–72, New York, NY, USA, 2006. ACM.
- [24] L. Zou and M. Godfrey. Detecting merging and splitting using origin analysis. In *Working Conference on Reverse Engineering*, pages 146–154, 2003.