On the Use of Clone Detection for Identifying Crosscutting Concern Code

Magiel Bruntink, *Student Member*, *IEEE Computer Society*, Arie van Deursen, *Member*, *IEEE*, Remco van Engelen, and Tom Tourwé, *Member*, *IEEE*

Abstract—In systems developed without aspect-oriented programming, code implementing a crosscutting concern may be spread over many different parts of a system. Identifying such code automatically could be of great help during maintenance of the system. First of all, it allows a developer to more easily find the places in the code that must be changed when the concern changes and, thus, makes such changes less time consuming and less prone to errors. Second, it allows the code to be refactored to an aspect-oriented solution, thereby improving its modularity. In this paper, we evaluate the suitability of clone detection as a technique for the identification of crosscutting concerns. To that end, we manually identify five specific crosscutting concerns in an industrial C system and analyze to what extent clone detection is capable of finding them. We consider our results as a stepping stone toward an automated "aspect miner" based on clone detection.

Index Terms—Clone detection, reverse engineering, aspect-oriented programming, crosscutting concerns, aspect mining.

1 Introduction

The tyranny of the dominant decomposition [1] implies that, no matter how well a software system is decomposed into modular units, some functionality (often called a *concern*) *crosscuts* the decomposition. In other words, such functionality cannot be captured cleanly inside one single module and, consequently, its code will be spread throughout other modules.

From a maintenance point of view, such a crosscutting concern is problematic. Whenever this concern needs to be changed, a developer should identify the code that implements it. This may possibly require him to inspect many different modules, since the code may be scattered across several of them. Moreover, identifying the code specifically related to the relevant concern may be difficult. Apart from the fact that the developer may not be familiar with the source code, this code may also be tangled with code implementing other concerns, again due to crosscutting. It should thus come as no surprise that identifying crosscutting code may be a time-consuming and error-prone activity, as shown by Soloway et al. for delocalized plans [2].

Aspect-oriented software development (AOSD) has been proposed for solving the problem of crosscutting concerns. Aspect-oriented programming languages have an abstraction mechanism targeted specifically at crosscutting concerns, called an *aspect*. This mechanism allows a developer to capture crosscutting concerns in a localized way.

In order to use this new feature and make the code more maintainable, existing applications written in ordinary programming languages could be evolved into aspect-oriented applications. Once again, this requires identifying the crosscutting concern code such that it can be refactored using aspects. The activity of finding opportunities for the use of aspects in existing systems is typically referred to as *aspect mining* [3].

To support developers in these tasks, some form of automation is highly desirable. Clone detection techniques are promising in this respect, due to two likely causes of code cloning occurring within scattered crosscutting concern implementations. First, by definition, scattered code is not well modularized. Several reasons can be identified for this lack of modularity, including missing features of the implementation language (exception handling or aspects, for instance), or the way the system was designed. In both cases, developers are unable to reuse concern implementations through the language module mechanism. Therefore, they are forced to write the same code over and over again, typically resulting in a practice of copying existing code and adapting it slightly to their needs [4].

Second, they may use particular coding conventions and idioms to implement *superimposed* functionality, i.e., functionality that should be implemented in the same way everywhere in the application. Logging and tracing are the prototypical examples of such superimposed functionality.

We hypothesize from these observations that clone detection techniques might be suitable for identifying some kinds of crosscutting concern code since they automatically

- *M. Bruntink, A. van Deursen and T. Tourwé are with the Department of Software Engineering, CWI, Kruislaan 413, 1098 SJ, Amsterdam, The Netherlands.*
  E-mail: {Magiel.Bruntink, Arie.van.Deursen, Tom.Tourwe}@cwi.nl.
- *A. van Deursen is also with the Faculty of Electrical Engineering, Mathematics and Computer Science, (EEMCS) at Delft University of Technology, Mekelweg 4, 2628 CD, Delft, The Netherlands.*
- *R. van Engelen is with Software Systems Development, ASML Netherlands B.V., De Run 6501, 5504 DR Veldhoven, The Netherlands.*
  E-mail: Remco.van.Engelen@asml.com.
- *T. Tourwe is also with the Programming Technology Lab, Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussels, Belgium.*

Manuscript received 15 Mar. 2005; revised 16 Aug. 2005; accepted 26 Aug. 2005; published online 3 Nov. 2005.
Recommended for acceptance by Harman, Korel, and Linos.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0065-0305.

detect duplicated code in a system's source code. In this paper, we report on a case study in which we evaluate the suitability of three different clone detection techniques for identifying crosscutting concern code. We manually identify five crosscutting concerns and evaluate to what extent the crosscutting concern code is matched by three different clone detection techniques. The evaluation considers token, AST, and PDG-based clone detection techniques (see Section 2) and provides a quantitative comparison of their suitability.

The case study considers crosscutting concerns prevalent in the source base of ASML, a producer of lithography systems based in Veldhoven, The Netherlands. A domain expert has manually annotated occurrences of five crosscutting concerns (Error Handling, Tracing, NULL-value Checking, Range Checking, and Memory Error Handling) in a component consisting of 16,406 lines of C code. The complete source base consists of roughly 10 million lines of C code, of which at least 25 percent is estimated to be dedicated to crosscutting concerns (based on the results found for the component considered in the case study).

The paper is structured as follows: Section 2 discusses related work in the areas of clone detection and aspect mining. In Section 3, we describe the case study and the five different crosscutting concerns. Subsequently, in Section 4, we detail the approach used to evaluate the capability of clone detection to find these crosscutting concerns. In Section 5, we present and explain the results obtained, followed by a discussion in Section 6. Finally, the paper is concluded in Section 7.

2 Related Work

2.1 Clone Detection Techniques

Clone detection techniques aim at finding duplicated code, which may have been adapted slightly from the original. Several clone detection techniques have been described and implemented:

- *Text-based* techniques [5], [6] perform little or no transformation to the "raw" source code before attempting to detect identical or similar (sequences of) lines of code. Typically, white space and comments are ignored.
- *Token-based* techniques [7], [8] apply a lexical analysis (tokenization) to the source code and, subsequently, use the tokens as a basis for clone detection.
- *AST-based* techniques [9] use parsers to first obtain a syntactical representation of the source code, typically an abstract syntax tree (AST). The clone detection algorithms then search for similar subtrees in this AST.
- *PDG-based* approaches [10], [11] go one step further in obtaining a source code representation of high abstraction. Program dependence graphs (PDGs) contain information of a semantical nature, such as control and data flow of the program. Kommondoor and Horwitz [10] look for similar subgraphs in PDGs in order to detect similar code. Krinke [11] first augments a PDG with additional details on expressions and dependencies, and similarly applies an algorithm to look for similar subgraphs.
- *Metrics-based* techniques [12] are related to hashing algorithms. For each fragment of a program, the values of a number of metrics are calculated, which are subsequently used to find similar fragments.
- *Information Retrieval-based* methods aim at discovering similar high level concepts by exploiting semantic similarities present in the source code itself (including the comments) [13], [14].

An important application of clone detection is the improvement of source code quality by refactoring duplicated code fragments [15]. Several authors have proposed to use clone detection in this setting. Baxter et al. [9] search for opportunities for replacing clones with calls to a function that factors out the commonalities among the clones. Balazinska et al. [16] focus on analyzing differences among clones, and their contextual dependencies, in order to determine suitable refactoring candidates. Van Rysselberghe and Demeyer [17] compare three classes of clone detection techniques, i.e., line matching, parametrized matching, and metric fingerprinting with respect to refactoring the obtained clones. Removal of clones by refactoring is further studied by Fanta and Václav [18].

Other applications exist as well. Van Rysselberghe and Demeyer, for example, use a clone detection algorithm to study the evolution of a software system [19]. In particular, they try to distinguish *move method* refactorings that were applied when evolving one version of the software into another.

Following Walenstein [20], [21], clone detection adequacy depends on application and purpose. Finding crosscutting concerns is a completely new application area, potentially requiring specialized types of clone detection.

2.2 Aspect Mining

Although research on aspect mining is still in its infancy, several prototype tools have already been developed that support developers in identifying crosscutting code. These tools vary in accuracy and the level of automation that they offer.

The *Aspect Browser* [22] is a programming environment that provides *text-based mining*, which means it relies on string pattern-matching techniques to identify aspects. A developer specifies a regular expression that describes the code belonging to the aspect of interest and a color. The programming environment then identifies the code conforming to the regular expression and highlights it using the associated color in the source code editor. Three concern elaboration tools, including the Aspect Browser, are compared in a recent study by Murphy et al. [23]. This study shows that the queries and patterns are mostly derived from application knowledge, code reading, words from task descriptions, or names of files. Prior knowledge of the system or known starting points strongly affect the usefulness of the outcomes of the analysis.

The *Aspect Mining Tool* [3] is an extension of the Aspect Browser that introduces a combination of text-based and *type-based mining*. Type-based mining considers the usage of types within an application to identify crosscutting code. It appears to be a good complement to simple text-based mining, and the combination of the two ensures that far less false positives and false negatives occur.

The *Prism* tool [24] (an earlier version is called AMTEX [25]) in its turn extends the Aspect Mining Tool and, additionally, provides a *type ranking* feature and takes control flow information into account. The type ranking feature is based on the assumption that types that are used widely in the application are a good indicator of crosscutting code. Therefore, the tool ranks the types in the system according to their use. The tool also takes control-flow information into account to identify aspects: e.g., it considers the values involved in conditional branches and the code involved in accessing these values (assignments, method calls, etc.). If such code is not well localized and appears in many places in the application, it may be a very good candidate for an aspect.

Ettinger and Verbaere discuss a totally different approach to aspect mining that identifies entangled code based on input by the developer and disentangles it using program slicing and aspect-oriented techniques [26]. In other words, the developer points out a particular expression or statement and a tool automatically computes the corresponding slice. The code fragment computed in this way can then be extracted into an aspect.

Fully automated tools for aspect mining are also proposed in the literature. Breu and Krinke propose a tool that dynamically analyzes Java software to identify aspects [27]. To that end, program traces are generated and analyzed automatically. The idea is to detect particular patterns occurring in the trace, such as a call to a particular method `a` that is always followed by a call to method `b`, or a call to method `c` that always occurs inside a call to method `d`. Such patterns could point to before/after aspects.

Shepherd et al. present a tool that uses a clone detection algorithm based on a program dependence graph [28] representation of Java code. The tool identifies possible aspects fully automatically, focusing currently on a specific type of aspects that introduces code before function calls (i.e., *before* advices). Their approach seems capable of finding such aspects in Java code, though the authors report that evaluation of their findings has been difficult due to a lack of a reference set of desirable aspects. In our work, such a reference set (consisting of manual annotations) is exploited in the evaluation.

Other techniques uses formal concept analysis [29] or metrics [30] to find crosscutting concern code, and combinations of these techniques are proposed to combine the respective advantages and counter the disadvantages [31].

Traditionally, aspect-oriented programming techniques have been applied to object-oriented applications. The idea of applying it for improving the modularity of large-scale C programs is not new, however. Most notably, Coady et al. report on an experiment using aspect-oriented techniques to modularize the implementation of prefetching within page fault handling in the FreeBSD operating system kernel [32]. To that end, they make use of an aspect language tailored specifically to the C programming language called *AspectC*, which is currently under development. However, in their experiment, the crosscutting code is identified manually rather than automatically.

3 Case Study

3.1 Setup

In Section 1, we argued that the presence of crosscutting concerns in a system could be a cause for code duplication. The failure to properly modularize a crosscutting concern, due to missing language features (e.g., exception handling or aspects) or improper system design, leads to programmers being forced to reuse crosscutting concern code in an ad hoc fashion, i.e., by copy-paste-adapt. Over time, this practice may even become part of the development process of an organization when common code fragments find their way into manuals as conventions or idioms.

The objective of this case study is to evaluate the hypothetical relation between five known (annotated) crosscutting concerns and the duplication of code in a component of a real system written in C. In particular, the case study focuses on the question of how well the code of these crosscutting concerns is found by three clone detectors implementing different clone detection techniques.

Clone detectors are designed to find duplicated fragments of code, using a specific clone detection technique (see Section 2.1). However, for the purpose of this case study, a clone detector is regarded as a search algorithm for crosscutting concern code. Consequently, well-known performance measures can be used from the field of information retrieval [33] (also suggested by Walenstein and Lakhotia in [21]). First, *recall* is used to evaluate how much of the code of a crosscutting concern is found by a clone detector. Second, *precision* gives the ratio of crosscutting concern code to unrelated code found by the clone detector. Finally, *average precision* provides an aggregate measure of the performance of a clone detector over all recall and precision levels. These measures are defined in detail in Section 4.5.

3.2 Subject System

The software component selected for this case study is called *CC* and consists of 16,406 lines of C code.[1] It is part of the larger code base (comprising over 10 million lines of code) of ASML. *CC* is responsible for the conversion of data between several data structures and other utilities used by communicating components.

Developers working on *CC* have expressed the feeling that a disproportional amount of effort is spent implementing "boiler plate" code, i.e., code that is not directly related to the functionality the component is supposed to implement. Instead, much of their time is spent dealing with concerns like Error Handling and Tracing (explained below).

1. The line count (LC) used throughout this paper is defined as the number of lines, excluding completely blank lines.

TABLE 1
Line Counts of the Five Concerns in the *CC* Component

Concern	Line Count (%)	Files (%)	Functions (%)
MEMORY	750 (4.6%)	8 (73%)	43 (27%)
NULL	617 (3.8%)	9 (82%)	67 (42%)
RANGE	387 (2.4%)	7 (64%)	38 (24%)
ERROR	927 (5.7%)	11 (100%)	128 (82%)
TRACING	1501 (9.1%)	10 (91%)	110 (70%)

The CC component consists of 16,406 lines, in 11 files, and 157 functions.

This problem is not limited to just the component we selected; it appears in nearly the entire code base. Since the developers at ASML use an idiomatic approach to implement various crosscutting concerns in applicable components, similar pieces of code are scattered throughout the system. Clearly, significant improvements in code size, quality, and comprehensibility are to be expected if such concerns could be handled in a more systematic and controlled way.

Five crosscutting concerns within *CC* were considered in the case study:

- Memory Error Handling: dedicated handling of errors originating from memory management functions.
- NULL-value Checking: checking the values of pointer parameters of a function against NULL (indicative of a failed or missing memory allocation attempt).
- Range Checking: checking whether values of input parameters (other than pointers) are within acceptable ranges.
- Error Handling, which is responsible for roughly three tasks: the initialization of variables that will hold return values of function calls, the conditional execution of code depending on the occurrence of errors, and, finally, administration of error occurrences in a data structure.
- Tracing: logging the values of input and output parameters of C functions to facilitate debugging. Each C function is required to perform tracing at its entry and exit points.

All together, these concerns comprise 4,182 lines (25.5 percent) of the 16,406 lines of *CC*. The details are shown in Table 1. In tables throughout the paper, the concerns are referred to by the short-hands MEMORY, NULL, RANGE, ERROR, and TRACING, respectively.

For each concern, Table 1 also shows the number of files and functions that includes at least one line of the concern. The *CC* component consists of 11 files, containing 157 functions. Fig. 1 illustrates the scattered nature of the NULL-value Checking concern by highlighting the code fragments that implement it. The vertical bars represent the files of *CC* and, within each vertical bar, horizontal lines of pixels correspond to lines of source code within the file. Colored lines are part of the NULL-value Checking concern. The other concerns exhibit a similarly scattered distribution.

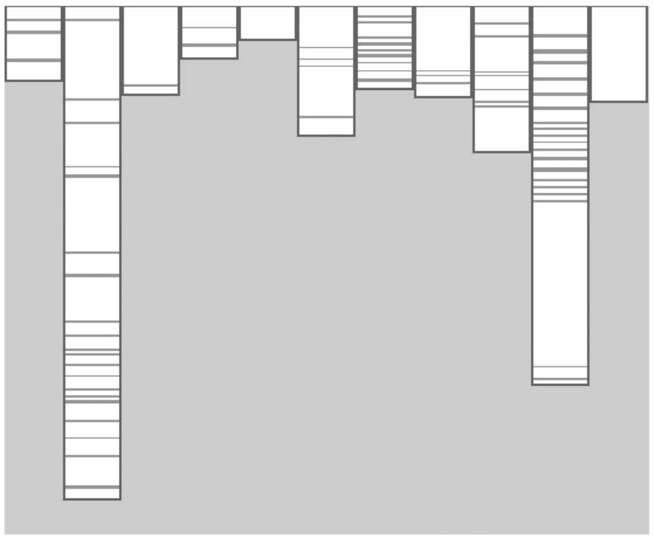

Fig. 1. Scattering of NULL-value Checking in the *CC* component. Vertical bars represent the .c files of *CC* (header files are not included). Within each vertical bar, horizontal lines of pixels correspond to lines of source code implementing the NULL-value Checking concern.

4 Experimental Setup

4.1 Annotation

The first phase of the case study consisted of a manual annotation effort performed by one of the authors of the *CC* component. For each of the concerns described in Section 3, the author of the component marked those source code lines which belong to each concern. Each line in the component was annotated with at most one mark and, thus, each source code line belongs to at most one of the concerns, or to no concern. See Table 1 in Section 3 for an overview of the number of lines belonging to each of the concerns.

4.2 Selected Clone Detectors

During the second phase of the case study, we performed clone detection on the *CC* component. For this purpose, we have used three different tools, which implement different clone detection techniques. First, we used the clone detection tool contained in Project Bauhaus (version 5.1.1) [34], a tool set developed at the University of Stuttgart with the goal to support program understanding and reengineering. The clone detector, called "ccdiml," is an implementation of a variation of Baxter's approach to clone detection [9] and, thus, falls in the category of AST-based clone detectors. Second, we used CCFinder (version 7.1.1) [7], a clone detection tool based on tokenized representations of source code. Finally, the PDG-based clone detector developed by Komondoor [10] was added to the study initially presented in [35]. This clone detector will be referred to as PDG-DUP throughout the paper.

A distinction between the clone detectors is due to the preprocessing required for Bauhaus' ccdiml and PDG-DUP before clone detection can commence. In effect, both of these tools detect clones in the preprocessed C code, instead of in the unpreprocessed code. In contrast, CCFinder is able to detect clones directly in the unpreprocessed code.

Consequently, Bauhaus' ccdiml and PDG-DUP have to process a larger amount of source code than the 16,406 lines mentioned earlier. The larger amount of source code may have performance implications. In total, the LC of the *CC* component after preprocessing is 40,005. On this particular component, the running time[2] of PDG-DUP is close to 4 hours of processor time on a 2 GHz AMD Athlon processor. In comparison, Bauhaus' ccdiml requires 3 minutes of processor time to calculate its results. CCFinder needs less than 1 minute to compute its clone classes, running on a slower 1.4 GHz Intel processor. It is not the goal of this study to compare the running times of the clone detectors, but repetition of the study on larger components may require the performance of PDG-DUP to be improved.

The use of preprocessing by Bauhaus' ccdiml and PDG-DUP has no major implications for the case study. Both Bauhaus' ccdiml and PDG-DUP create a mapping for clones detected in the preprocessed code back to the original unpreprocessed code. As a consequence, the clone detection results can be interpreted based on the unpreprocessed code.

4.3 Clone Detector Configuration

The selected clone detection tools can be configured prior to execution which affects the types of clones detected.

4.3.1 Bauhaus' ccdiml

A prevalent setting of clone detectors is the minimum size of the reported clones. For the purpose of this case study, the minimum size should be set such that the largest, still tractable (with respect to processing time and memory requirements) volume of results is obtained. In case of Bauhaus' ccdiml, the minimum clone size was set to be two lines. For other (larger) components, this value may have to be increased, such that a smaller number of (small) clones is obtained.

Furthermore, Bauhaus' ccdiml is capable of detecting three types of clones. First, *exact* clones are simply verbatim copies, although white space and comments are ignored. Second, *parametrized* clones are like exact clones but the leaves of the AST's are ignored during comparison. The result is that variable and type names and literal values are not taken into account. Third, *near* clones are like parametrized clones but allow for insertion and deletion of code. For our experiment, we consider only the first two types, i.e., exact and parametrized clones, because near clones cannot be abstracted into clone classes (see Section 4.4).

The exact command line (without input and output files) to execute ccdiml is given by:

```
ccdiml -all_statements -minlines 2
```

4.3.2 CCFinder

For CCFinder, we left all settings at their defaults, except for the minimum length a clone must have in order to be included in the output: A clone must at least be seven tokens long. A smaller minimum length resulted in more clones than could be handled by CCFinder, causing an abort.

2. Excluding parsing the C code and calculating the PDG.

CCFinder was executed using the following command line options:

```
ccfinder C -b 7,1.0
```

where `C` indicates that the tokeniser should expect C code, and `-b 7,1.0` sets the minimum clone size to seven tokens.

4.3.3 PDG-DUP

The size of clones detected by the PDG-based clone detector is expressed as the number of vertices in the PDG that are included in a clone. For the *CC* component, three vertices is the smallest minimum size that could still be handled; using a minimum size of two vertices caused PDG-DUP itself to abort.

Furthermore, PDG-DUP requires the user to set a commonality threshold, which is used to remove clones that are overlapped too much by other clones. Per the recommendation of PDG-DUP's author, the `COMMON` threshold was set to 80 percent.

4.4 Abstracting Clone Detection Results

Some clone detectors produce output consisting of pairs of clones, i.e., they report which pairs of code fragments are similar enough to be called clones. However, for our purpose, the pairs of clones are not very interesting. Instead, we investigate the groups of code fragments that are all clones of each other. These groups of code fragments are termed *clone classes* [7].

More formally, a clone detector defines a relation between code fragments and typically yields the tuples of this relation as its output. Instead of investigating these tuples on their own, we assume this *clone* relation to be an equivalence relation. It is clear that a clone fragment is always either an exact or parametrized clone of itself (reflexivity). Also, if code fragment A is an exact or parametrized clone of code fragment B, then it is clear that B is also an exact or parametrized clone of A (symmetry). Finally, if code fragment A is a clone of B and B is a clone of C, then A is also an exact or parametrized clone of C (transitivity). Subsequently, clone classes are comprised of the equivalence classes of the *clone* relation.

The output of CCFinder and PDG-DUP indeed describe equivalence relations between code fragments and, thus, obtaining the clone classes is a straightforward task. However, our version of Bauhaus' ccdiml does not produce an equivalence relation. Given the types of clones we include in the study, i.e., either exact or parametrized clones, it is justified to augment the output of ccdiml such that it does constitute an equivalence relation. For this purpose, we use grok, a relational algebra program developed by Holt [36]. The equivalence classes were obtained by applying a simple union-find algorithm to the reflexive transitive closure of the clone relation.

4.5 Measurements

In the third phase of the case study, we performed measurements to test the hypothesis that the clone classes detected by the three clone detectors, i.e., Bauhaus' ccdiml, CCFinder, and PDG-DUP, match the annotated crosscutting concern code.

TABLE 2
Raw Clone Detection Results

	Clones	Clone Classes	LC
Bauhaus	5,694	617	8,606
CCFinder	8,105	1,101	10,584
PDG-DUP	23,427	4,240	8,292

A clone class defines a (noncontiguous) region of source code that is related according to a clone detector. The manually annotated source code is also partitioned in several (noncontiguous) regions, namely, those lines of source code that implement the Memory Error Handling, NULL-value Checking, Range Checking, Error Handling and Tracing concerns, and other code. With regard to the goal, an interesting criterion for evaluation is the extent to which the regions defined by the annotations are matched by the regions defined by the clone classes.

Section 3.1 proposed to use performance measures from the field of information retrieval to evaluate the match between crosscutting concern code and clone classes. We now define those measures in detail:

Definition 1 (Concern). *Each* concern *is represented by a set containing the source lines of the concern, as specified by the annotations.*

Definition 2 (Clone). *A* clone *is defined as a set of source code lines.*

Definition 3 (Clone Class). *A* clone class *is a set consisting of clones. For a clone class CC, we define*

$$\mathrm{lines}(CC) = \bigcup_{c \in CC} c.$$

Definition 4 (Clone Class Collection). *A* clone class collection *is a set of clone classes. For a clone class collection D, we also define*

$$\mathrm{lines}(D) = \bigcup_{d \in D} \mathrm{lines}(d).$$

Definition 5 (Recall and Precision). *Let C be a concern, and D a clone class collection; then, we define* recall *and* precision *[33] as r and p, respectively:*

$$r(C, D) = \frac{|C \cap \mathrm{lines}(D)|}{|C|},$$
$$p(C, D) = \frac{|C \cap \mathrm{lines}(D)|}{|\mathrm{lines}(D)|},$$

where $|S|$ is the cardinality of a set S. Clearly, $0 \leq r \leq 1$ and $0 \leq p \leq 1$.

Originally, some blank lines and lines containing only opening and closing brackets, i.e., "{" and "}", were included in the annotations. Such lines will never be included in the results of the PDG-DUP clone detector because such lines are not included in PDG-DUP's mapping from PDG vertices to source code. Therefore, such lines had their annotations removed.

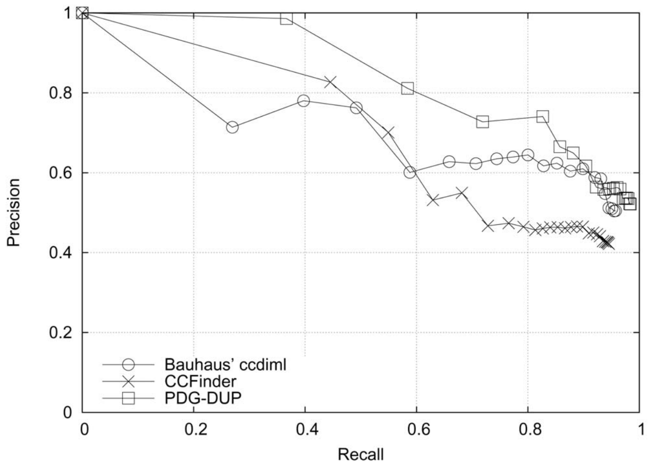

Fig. 2. Recall-Precision graphs for Memory Error Handling. Twenty-one clone classes are shown for Bauhaus' ccdiml, 26 for CCFinder, and 20 for PDG-DUP.

Note that this raises an issue of fairness. While PDG-DUP does not include such lines in its results, Bauhaus' ccdiml and CCFinder possibly do. Since those lines no longer belong to a concern (have an annotation), the precision of Bauhaus' ccdiml and CCFinder may be adversely affected. Therefore, blank lines and lines containing only opening and closing brackets were also removed from the clones classes calculated by Bauhaus' ccdiml and CCFinder. The line counts presented in Table 1 and Table 2 were performed after the removal of these lines. We are not aware of other issues regarding fairness at the syntactical level.

The clone detectors produce a large number of results for CC. See Table 2 for an overview of these results. Given the raw clone detection results, many clone class collections are possible for each clone detector. For the comparion of the clone detectors, we consider appropriate selections of clone classes instead. First, we define the notion of a clone class selection:

Definition 6 (Clone Class Selection). *Given a clone class collection D, a* clone class selection *S_k is a sequence of clone classes $\langle x_1, x_2, \ldots, x_n \rangle$, such that $\{x_1, x_2, \ldots, x_k\} \subseteq D$.*

For each clone detector and each concern, we consider a clone class selection of size k, i.e., we select k clone classes from the set of all clone classes found by a clone detector. Given such a selection, a recall-precision graph [33] can be plotted to give an overview of the quality of the match between the selected clone classes and a concern. For example, Fig. 2 contains the recall-precision graphs for the match between the three clone detectors and the Memory Error Handling concern. A recall-precision graph shows the recall (x-axis) and precision (y-axis) levels obtained for a clone class selection S_k. Each point on a recall-precision graph shows the recall and precision of a clone class collection $\{x_1, x_2, \ldots, x_l\}$ consisting of the first l clone classes of S_k and $1 \leq l \leq k$.

For each concern, we attempt to find a clone class selection such that the selected clone classes together provide the best possible match with a concern. Intuitively, good matches are provided by clone class selections that result in high recall while maintaining high precision. The *average precision* (AP) [33] is a commonly used measure that captures this intuition. We define average precision (AP) [33] for a clone class selection as follows:

Definition 7 (Average Precision). *Given a clone class selection $S_k = \langle x_1, x_2, \dots, x_k \rangle$, let S'_l ($1 \le l \le k$) be the clone class collection $\{x_1, x_2, \dots, x_l\}$ consisting of the first l clone classes in the selection S_k. With C a concern, we now define:*

$$AP(C, S_k) = \sum_{i=1}^{k} p(C, S'_i)\Delta r(C, S'_i),$$

where $\Delta r(C, S'_i)$ is the difference between $r(C, S'_i)$ and $r(C, S'_{i-1})$ ($S'_0 = \emptyset$).

Each clone detector yields clone classes containing the clones it found in *CC*. To compare the clone detectors at matching crosscutting concern code, for each concern and each clone detector, a clone class selection is made such that the average precision for the selection is (approximately) optimal.

Additionally, the average precision measure is helpful for the comparison of the recall-precision graphs obtained for the clone detectors, since average precision maps each recall-precision graph to a single number. Therefore, average precision is used as the primary means of evaluation for the performance of the clone detectors at matching the various concerns.

4.6 Calculating Clone Class Selections

Unfortunately, finding a clone class selection S_k that has an optimal average precision is a computationally hard problem. Processing the results has therefore been done using an approximate (greedy) algorithm which iterates k times over all clone classes, and each iteration selects the clone class which adds the most average precision. Previously selected clone classes and lines of a concern are disregarded, such that each iteration considers those lines of a concern that are still remaining.

See Algorithm 1 for a pseudocode specification of the algorithm. It computes a clone class selection of size k as follows: Line 1 initializes `remainder` with all the lines of the specified concern. `remainder` will be used as the work list for the algorithm. The loop initiated in line 6 makes sure that k clone classes will be selected. In line 10, the iteration over all clone classes is started. Lines 11-12 calculate the concern lines included in the clone class under consideration (`hits`), and the other lines that the clone class includes (`misses`). Based on the hits and misses, the algorithm calculates the current precision (`P`), added recall (`dR`), and added average precision (`dAP`) in lines 14-18. Subsequently, the added average precision is then compared to the current maximum and, if greater, the current clone class is marked as the current best choice (lines 20-25). After the iteration over all clone classes is finished, the clone class that adds the most average precision is known. The hits and misses of this clone class are added to the totals (lines 28-29), and the hits are subtracted from the remainder of the concern (line 30). In line 31, the selection is extended with the best clone class, after which the algorithm iterates in order to determine the next best clone class for the ordered selection. Finally, the selected clone class is written to output at line 34.

Algorithm 1: Select Clone Classes

```
Select Clone Classes(concern, cloneClasses, k)
1. remainder $\leftarrow$ concern
2. totalHits $\leftarrow \emptyset$
3. totalMisses $\leftarrow \emptyset$
4. selection $\leftarrow \langle\rangle$

6.  for i = 1 to k
7.    bestCC $\leftarrow \emptyset$
8.    bestdAP $\leftarrow 0$

10.   for each CC $\in$ cloneClasses
11.     hits $\leftarrow$ remainder $\cap$ CC
12.     misses $\leftarrow$ CC $-$ hits $-$ totalMisses

14.     tHits $\leftarrow$ totalHits $\cup$ hits
15.     tMisses $\leftarrow$ totalMisses $\cup$ misses
16.     P $\leftarrow \sharp$(tHits)/($\sharp$(tHits) + $\sharp$(tMisses))
17.     dR $\leftarrow \sharp$(hits)/$\sharp$(concern)
18.     dAP $\leftarrow$ P $\cdot$ dR

20.     if dAP > bestdAP
21.       bestCC $\leftarrow$ CC
22.       bestdAP $\leftarrow$ dAP
23.       bestHits $\leftarrow$ hits
24.       bestMisses $\leftarrow$ misses
25.     end if
26.   end for

28.   totalHits $\leftarrow$ totalHits $\cup$ bestHits
29.   totalMisses $\leftarrow$ totalMisses $\cup$ bestMisses
30.   remainder $\leftarrow$ remainder $-$ bestHits
31.   selection $\leftarrow$ APPEND(bestCC, selection)
32. end for

34. OUTPUT(selection)
```

The algorithm described above possibly calculates clone class selections that are not optimal with respect to their average precision. Small differences between average precision values are therefore required to be interpreted cautiously.

5 Results

For each of the five crosscutting concerns, we used Algorithm 1 to calculate a clone class selection for each of the clone class collections yielded by the clone detectors. The maximum number (k) of clone classes to select per clone class collection was set to 100. In case of the *CC* component, this value is high enough to ensure that the point is reached where selecting an additional clone class no longer adds average precision. The recall-precision graphs presented for each concern are limited to the range where average precision is added by selecting each new clone class. The average precision levels reached by the clone class selections are presented in Table 3.

TABLE 3
Average Precision Values

	MEMORY	NULL	RANGE	ERROR	TRACING
Bauhaus	.65	.99	.71	.38	.62
CCFinder	.63	.97	.59	.36	.57
PDG-DUP	.81	.80	.42	.35	.68

Figs. 2, 4, 6, 8, and 10 depict the recall-precision graphs of the clone class selections made for the five crosscutting concerns. All graphs are rooted at 0.0 recall and 1.0 precision, which is the case where no clone classes are selected, i.e., S_0. The results for each of the concerns will now be discussed in detail.

5.1 Memory Error Handling

Based on the recall-precision graphs, and the resulting average precision values (see Table 3) for Memory Error Handling, the PDG-DUP clone detector clearly performs best. The recall-precision graph for PDG-DUP is significantly above those of Bauhaus' ccdiml and CCFinder for almost all l. Consequently, the final AP level reached by PDG-DUP is significantly higher as well.

The difference between Bauhaus' ccdiml and CCFinder is not so clear. Bauhaus' ccdiml does better in the high recall area (above .60 recall, in the right half of Fig. 2), while CCFinder does better in the low recall area. Their respective AP values are quite close as well.

Observe in Fig. 2 that CCFinder reaches .45 recall using only one clone class (the first data point for CCFinder). This particular clone class contains 96 clones which are six lines in length. Fig. 3 shows an example clone from this class. While the lines marked with "M" belong to the Memory Error Handling concern, only the lines marked with "C" are included in the clones. Note that completely blank lines and lines containing only brackets have no annotation in this example, as was discussed earlier in Section 4.1. Consequently, those lines were also removed from the clone classes to allow for a fair comparison. In Fig. 3, the lines which were removed from the clones are marked with "-".

As can be seen from the line markers, the CCFinder clone captures the Memory Error Handling fragment only partly, stopping half way the parameter list of a function call. CCFinder allows clones to start and end with little regard to syntactic units (see Fig. 12 for another example). In contrast, Bauhaus' ccdiml does not allow this, due to its AST-based clone detection algorithm. PDG-DUP is

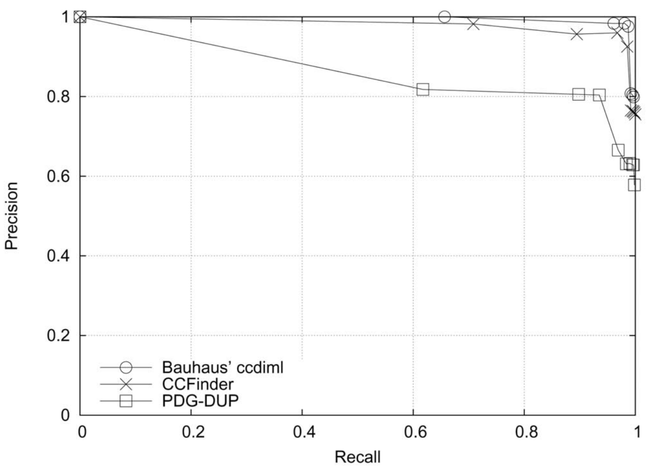

Fig. 4. Recall-Precision graphs for NULL-value Checking. Eight clone classes are shown for Bauhaus' ccdiml, 10 for CCFinder, and 10 for PDG-DUP.

bound to conform to language syntax as well since its PDGs are built on top of ASTs.

The first clone class of CCFinder does not cover Memory Error Handling code exclusively. In Fig. 2, note that the precision obtained for the first clone class is roughly .83. Through inspection of the code, we found that some of the clones do not cover memory error handling code at all, but code that is similar at the lexical level, yet conceptually different. In other words, some clones capture entirely different functionality. The results show that PDG-DUP is better able to make this distinction, resulting in a higher level of precision.

5.2 NULL-Value Checking

Fig. 4 shows the recall-precision graphs for the NULL-value Checking concern. All clone detectors obtain excellent results here, with Bauhaus' ccdiml and CCFinder even approaching 1 (perfect) average precision. PDG-DUP performs significantly worse than both Bauhaus' ccdiml and CCFinder, but still obtains a high average precision of .80.

The clone class that was selected first in the case of Bauhaus' ccdiml captures .66 of the concern at a precision of 1. This class consists of 77 clones, spanning 405 lines of NULL-value Checking code. In Fig. 5, we show an example clone of this clone class. The lines marked with "N" belong to the NULL-value Checking concern and those marked with "C" are the lines included in the clone. Again, lines

```
M C if (r != OK)
  - {
M C   ERXA_LOG(r, 0, ("PLXAmem_malloc failure."));
  -
M C   ERXA_LOG(CCXA_MEMORY_ERR, r,
M C          ("%s: failed to allocated %d bytes.",
M            func_name, toread));

M     r = CCXA_MEMORY_ERR;
    }
```

Fig. 3. CCFinder clone ("C" lines) covering Memory Error Handling ("M" lines).

```
N C if ((r == OK) && (msg == (void *) NULL))
  - {
N C   r = CCXA_PARAMETER_ERR;
  -
N C   ERXA_LOG(r, 0,
N C     ("%s: input parameter '%s' is NULL.",
N C     func_name,
N C     "msg"));
  - }
```

Fig. 5. Bauhaus' ccdiml clone ("C" lines) covering NULL-value Checking ("N" lines).

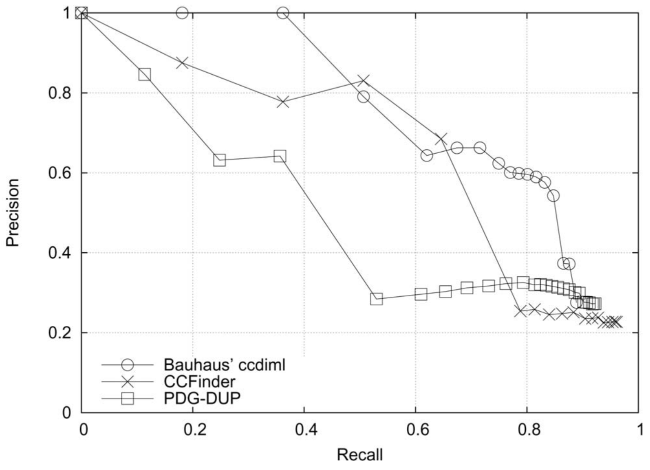

Fig. 6. Recall-Precision graphs for Range Checking. Seventeen clone classes shown for Bauhaus' ccdiml, 18 for CCFinder, and 26 for PDG-DUP.

marked with "-" are included in the original clones, but were removed for the comparison.

The coding style adopted at ASML distinguishes three kinds of pointer parameters: input, output, and output pointer parameters. All parameters of type pointer are checked against NULL by the NULL-value checking concern. Both input and output parameters are checked in the same way, whereas a check for an output pointer parameter differs slightly from the other checks as it requires an extra dereference. The clone detection results confirm this; the first two selected clone classes of Bauhaus' ccdiml cover input/output and output pointer parameter checks with high precision, respectively.

The first clone class selected for PDG-DUP results in far lower precision than Bauhaus' ccdiml or CCFinder. It turns out that, although the first clone classes are similar for all clone detectors, all the clones of PDG-DUP are extended with additional lines. For instance, the fragment in Fig. 5 is also found by PDG-DUP, but as part of a larger clone. The larger clone includes the declarations (and initializations) of the `r` and `func_name` variables, which are not considered to be part of the NULL-value Checking concern. The purpose of these variables is to facilitate error handling in general and, thus, their declarations are part of the Error Handling concern.

PDG-DUP adds these declarations to its clones despite the fact that the declarations are not textually near the other code fragment in Fig. 5. This behavior is due to the existence of data dependency edges in the PDG between nodes representing uses of the variables and nodes representing their declarations. Since Bauhaus' ccdiml and CCFinder do not regard data dependencies, they do not extend their clones to include the declarations.

5.3 Range Checking

As indicated by the average precision values, Bauhaus' ccdiml (AP .71) outperforms the other clone detectors at finding Range Checking code. Especially PDG-DUP performs badly, resulting in only .42 average precision. CCFinder's performance is in between Bauhaus' ccdiml and PDG-DUP, with .59.

```
    default:
R C   r = CCXA_PARAMETER_ERR;

R C   ERXA_LOG(r, 0,
R C     ("%s: unknown type code encountered (%d).",
R C     func_name,
R C     desc_src->type_code));
```

Fig. 7. PDG-DUP clone ("C" lines) covering Range Checking ("R" lines).

The recall-precision graph in Fig. 6 shows some significant drops in precision. For example, the fourth clone class selected for PDG-DUP (recall .53, precision .28) causes a drop in precision of .36. Similar drops in precision happen for Bauhaus' ccdiml (clone classes 3 and 4) and CCFinder (clone class 5). These clone classes add code fragments which are similar to the one in Fig. 7, yet do not represent Range Checking code. In fact, these fragments perform checks on the return values of function calls, while Range Checking is concerned with checking the values of input parameters. The way invalid values are handled is identical in both cases, which explains why these fragments are included in clone classes together with Range Checking code.

5.4 Error Handling

Of the concerns we consider, error handling is clearly the worst in terms of the recall-precision graphs (Fig. 8) and average precision. Bauhaus' ccdiml has a slight advantage over CCFinder and PDG-DUP in the low recall area.

The error handling concern can be partitioned into three subconcerns: *initialization*, *error linking*, and *skipping*. First, the *initialization* subconcern deals with the initialization of variables used to keep track of return values. Second, *error linking* handles the administration of error occurrences in a data structure. Third, *skipping* is concerned with making sure that specific parts of a function are not executed in case an error has occurred.

Further inspection has shown that the initialization and error linking subconcerns are included almost entirely by

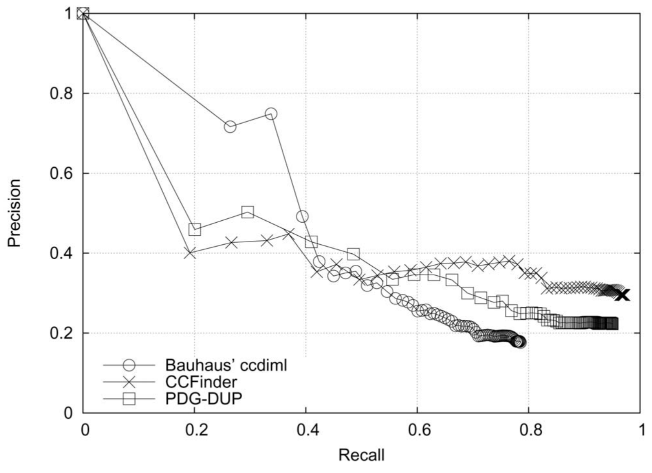

Fig. 8. Recall-Precision graphs for Error Handling. Sixty-nine clone classes shown for Bauhaus' ccdiml, 62 for CCFinder, and 74 for PDG-DUP.

```
S if (r == OK)
  {
    r = FD_read(read_fd,
                &msg_hdr,
                (int) sizeof(CCCN_msg_header));
  }
```

Fig. 9. Instance of the skipping concern (“S” line).

the first and second clone class of Bauhaus’ ccdiml, respectively. However, the skipping subconcern is found very badly, which explains why the error handling concern in general is found badly.

Consider the code fragment in Fig. 9, a simple example of code belonging to the skipping subconcern. The line marked with “S” belongs to the skipping (sub)concern. Skipping accounts for 445 lines of code, i.e., 2.7 percent of the CC component and, furthermore, it is present in the entire code base. In the example, the `r` variable is used to hold return values of previous function calls and the if statement ensures the conditional execution of the remaining code.

The clone classes yielded by the three different clone detectors do not provide a good match with this concern for the same reason: The pieces of skipping code are simply too small to qualify for clones by themselves due to the limits we set in Section 4. Furthermore, the code that appears inside the `if` statements can differ greatly. As a result, no clone classes are found that cover just the skipping concern. However, some clone classes cover the skipping subconcern by accident, i.e., the clones cover a large number of nonconcern lines compared to the number of skipping lines covered.

5.5 Tracing

The average precision values show that PDG-DUP clone classes have the best match with Tracing code, although the difference with Bauhaus’ ccdiml is not large (see Fig. 10). All three clone detectors show notable drops in precision; Bauhaus’ ccdiml at clone class 3, CCFinder at clone class 2, and PDG-DUP at clone class 5.

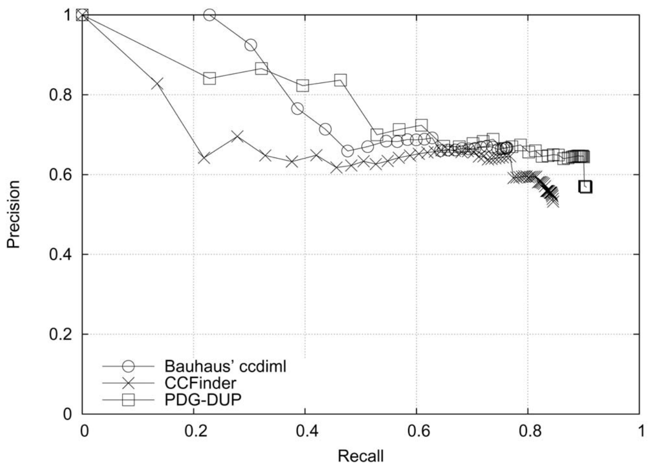

Fig. 10. Recall-Precision graphs for Tracing. Twenty-eight clone classes shown for Bauhaus’ ccdiml, 65 for CCFinder, and 41 for PDG-DUP.

```
T C THXAtrace(CC,
T C           THXA_TRACE_INT,
T C           func_name,
T C           "< () = %R",
T C           r);
```

Fig. 11. Bauhaus’ ccdiml clone (“C” lines) covering Tracing (“T” lines).

It turns out that these drops in precision are caused by a small number of functions which are almost entirely cloned. The clone classes mentioned before contain clones which are almost as big as entire functions. As a result, the included Tracing code is accompanied by a large number of lines belonging to other concerns.

The first selected clone class for Bauhaus’ ccdiml obtains .23 recall at 1.0 precision. An example clone from the first clone class is shown in Fig. 11. In total, 71 clones of this class are present in the CC component, spanning 343 lines. The lines belonging to the Tracing concern are marked with “T,” while lines marked with “C” belong to the example clone.

The clones for PDG-DUP contain non-Tracing lines for the reason also observed for the NULL-value Checking concern. Due to a data dependency between the use of the `func_name` variable in the call to the tracing function and the declaration (and initialization) of `func_name`, the declaration is also included in the clone. Again, this declaration is not part of the Tracing concern, but instead it is included in the Error Handling concern.

All the code belonging to the Tracing concern is very similar to the example in Fig. 11. The `THXAtrace(...)` function is always called, and its first three arguments are typically the same. However, a variable number of arguments can follow the first three. As a consequence, we also find clone classes which consist of calls to `THXAtrace(...)` with five arguments, six arguments, and so on. In fact, the second clone class selected for Bauhaus’ ccdiml contains clones of the `THXAtrace(...)` function call with six arguments.

CCFinder does not yield clones of the `THXAtrace(...)` function call with less than six arguments simply because we have limited the minimum size of a clone to seven tokens (see Section 4). However, clone classes including `THXAtrace(...)` function calls with less than six arguments do turn up, but those also include a number of non-Tracing lines. A clone belonging to the first clone class selected for CCFinder is shown in Fig. 12. It does, in fact, include many of the same lines as the first clone class selected for Bauhaus’ ccdiml, but, as can be seen in Fig. 12, the clones are extended with non-Tracing lines.

```
T C THXAtrace(CC,
T C           THXA_TRACE_INT,
T C           func_name,
T C           "< () = %R",
T C           r);
  -
  C return r;
  - }
```

Fig. 12. CCFinder clone (“C” lines) covering Tracing (“T” lines).

TABLE 4
Average Precision Values for Combined Clone Detection Results

	MEMORY	NULL	RANGE	ERROR	TRACING
Bauhaus ∪ CCFinder	.69	.98	.76	.52	.70
Bauhaus ∪ PDG-DUP	.83	.99	.72	.53	.78
CCFinder ∪ PDG-DUP	.79	.97	.62	.38	.73
Bauhaus ∪ CCFinder ∪ PDG-DUP	.77	.98	.79	.54	.81

5.6 Combining Clone Detectors

Table 4 contains average precision values obtained for combinations of the three clone detectors. A combination of two (or more) clone detectors consists of the union of their respective clone class collections. The union is then subject to the same selection procedure as for the individual clone detectors, i.e., Algorithm 1. The resulting clone class selections then possibly consist of a mix of clone classes from the combined clone detectors.

Some combinations of clone detectors perform better than the individual clone detectors at matching some concerns. The Range Checking concern is matched better by the combination of Bauhaus' ccdiml and CCFinder than by any individual clone detector (see Tables 3 and 4). The combination of all clone detectors reaches the highest AP for the Range Checking concern. The same result is true for the Error Handling and Tracing concerns. Matching of Error Handling code especially seems to benefit from combining clone detectors.

Clearly, combinations of clone detectors allow for the balancing of the weaknesses and strengths of the individual clone detectors. Further research will be required to provide a qualitative explanation of these results.

It is expected that the combination of all clone detectors performs best at matching a concern. However, this is not the case for the Memory Error Handling and NULL-value Checking concerns. For Memory Error Handling, the AP for Bauhaus ∪ PDG-DUP is .83 (see Table 4), while, for Bauhaus ∪ CCFinder ∪ PDG-DUP, the obtained AP is lower at .77. Similarly, for NULL-value Checking, the AP for Bauhaus ∪ PDG-DUP is .99, while Bauhaus ∪ CCFinder ∪ PDG-DUP falls short at .98. These small anomalies can be explained by the fact that a nonoptimal algorithm is used to calculate the clone class selections.

5.7 Summary

Based on the average precision values in Table 3, the clone class selections obtained for Bauhaus' ccdiml provide the best match with the Range Checking, NULL-value Checking, and Error Handling concerns. However, CCFinder's clone class selections perform almost equally well for NULL-value Checking and Error Handling.

For the remaining concerns, i.e., Tracing and Memory Error Handling, the clone class selections for PDG-DUP perform best. Especially for NULL-value Checking, high average precision values are obtained, which even approach the perfect score in case of Bauhaus' ccdiml and CCFinder. The Error Handling concern is matched badly across the board, however.

Combining clone detectors is expected to improve the matching of crosscutting concern code, and our results confirm this expectation (except for some small anomalies due to the algorithm used in the evaluation). Especially the Error Handling and Tracing concerns are matched better when a combination of clone detectors is considered.

6 Discussion

6.1 Limitations

Clone detection techniques identify the code of our crosscutting concerns because code duplication is the way programmers at ASML reuse (parts) of these concerns. Furthermore, the development process at ASML has a strong idiomatic character. First, it provides strict rules on the implementation of the concerns in question, in the sense that programmers are required to implement the concerns for (almost) every function. Second, programmers are inclined to implement the crosscutting concerns in a similar fashion each time, maybe even making verbatim copies of existing implementations. The programming manual used by each programmer even provides templates for the implementation of some crosscutting concerns, such as Error Handling, Tracing, and NULL-value Checking. As a result, a large number of similar implementations of the crosscutting concerns are scattered across the system. We view an idiomatic nature of the development process (such as the one at ASML) as a major condition to the applicability of our results.

The evaluation performed does not punish the clone detectors for yielding irrelevant clone classes. For example, clone classes that do not contain any lines of a concern are not considered. In general, the clone class selection algorithm only considers those clone classes that can add recall at a given point during the selection. If no such clone classes remain, then the average precision of the clone class selection is fixed. Clone classes which do not add recall do not influence the average precision since their Δr is 0 by definition. For the evaluation of the case study hypothesis, it does not matter that clone detectors also yield irrelevant clone classes: The case study addresses the question which clone detector is capable of providing the closest match between crosscutting concern code and the detected clones, not whether all detected clones match crosscutting concern code.

A limitation that surfaced mainly for the PDG-DUP clone detector is due to the line granularity of the case study. Each source code line can belong to at most one concern, while, in some cases, we could consider including a line in multiple concerns. An example was discussed in Section 5.2. In that case, the first PDG-DUP clone class for the NULL-value

Checking concern consists of clones covering mostly NULL-value Checking code. However, each clone is extended with the declarations of the variables used within the clones. In turn, these declarations are considered to be part of another concern, Error Handling in this case. The main use of these variables lies with the Error Handling concern, yet they are used in an auxiliary fashion by a couple of other concerns, e.g., NULL-value Checking, Tracing, and Memory Error Handling. An appropriate solution could be to allow lines to belong to multiple concerns, however, this option remains unexplored for now.

6.2 Oracle Reliability

A key element of the case study consists of source code annotations produced by a human oracle. The main author of the studied component marked those lines of source code that are to be considered part of one of five (crosscutting) concerns. Several measures were employed to assure the quality of these annotations.

First, the annotated source lines were manually inspected to identify any obvious mistakes made during annotation. As a result, annotations of blank lines and lines containing nothing but opening and closing brackets were removed (see Section 4.1).

Second, the crosscutting concerns considered in the case study are not specific to the selected component. In fact, all five concerns are present in a large number of other components of the ASML source base. Furthermore, concerns such as error handling, tracing, and NULL-value checking are described in detail by the actual programming manuals. As a consequence, the nature of the crosscutting concerns was well-known, allowing the annotations to be checked for nontrivial mistakes.

Third, clone classes that exhibit high added recall fractions, yet cause significant drops in precision, were inspected manually and discussed with the component's author. Clone classes such as these relate a number of clones that match part of a concern (explaining the added recall) to a large number of clones that match other functionality (explaining the drops in precision). Especially when a number of clones matches a part of the concern precisely and the other clones in the clone class match only other functionality, doubts about the completeness of the annotations could arise. An example of such a clone class was encountered for the Range Checking concern, as discussed in Section 5.3. The author of the component verified that the nonconcern clones were in fact implementing different functionality. No missing annotations were discovered in this way.

6.3 Consequences for Aspect Mining

The results presented in this paper have consequences for the extent to which the studied clone detectors can be used for the purpose of automatic aspect mining. In the case study, we determined the (approximately) best match of (the clone classes of) each clone detector with five crosscutting concerns. We were able to determine those matches due to the availability of annotations that map each source line to either a crosscutting concern or to other functionality.

Automatic aspect mining is expected to work without manually obtained annotations. An aspect miner based on a clone detector typically fits the following framework. First, the clone detector calculates clone classes. Second, a clone class selection is made such that best aspect candidates are selected first. The absence of annotations requires that a different approach is used to perform the clone class selection.

It is reasonable to believe that an automatic aspect miner is not going to deliver a better match with crosscutting concerns than a manual annotation effort. In that sense, the clone class selections we obtained based on the annotations can be seen as the best result that can be expected of an automatic clone class selection approach. For example, the results show that the Error Handling concern is matched badly by the clone class selections that we calculated based on the annotations (see Section 5.4). Therefore, an automatic aspect miner that uses one of the three clone detectors studied here cannot be expected to provide a good match with the Error Handling concern. In general, the average precision values in Tables 3 and 4 give an indication of the suitability of the three clone detectors for the purpose of automatically mining for aspect candidates like the five crosscutting concerns considered in the case study.

An example aspect mining approach using AST-based clone detection is described in [37]. Clone classes can be characterized by simple metrics like the number of clones contained in the class or the number of lines covered by the class, but more complex metrics can be derived as well, such as the distribution of clones over different files. These metrics can subsequently be used to guide the clone class selection process.

6.4 Clone Extension

The results show many examples of clone classes that consist of clones which all include some lines of a particular concern, yet also some other lines. One example is the first selected PDG-DUP clone class for the NULL-value Checking concern (see Section 5.2). In that case, each clone has been extended to include the variable declarations of the variables used in the clone. These variable declarations are not considered to be part of the NULL-value Checking concern and, hence, their inclusion results in a lower precision. Another example was discussed for the Tracing concern, where the clones of the first selected clone class for CCFinder include the `return` statement which always follows the tracing code at the end of a function.

The clone detectors considered in this study are programmed such that (only) maximally large clones are presented to the user. Smaller clones which appear as intermediate results are removed when they are (partly) "subsumed" by bigger clones and, hence, do not show up in the final results. For the purpose of finding duplicated code, this is desirable behavior. However, it is clear from the examples above that, in case of matching crosscutting concerns, precision can be adversely affected. If the subsumed clone classes had not been discarded from the final results, those classes would have been selected instead of the current ones, resulting in higher precision.

It should be noted that failing to discard subsumed clones altogether will probably result in an intractable number of results. Instead, a better solution is to allow the user to control the extent to which the clone detector discards subsumed clones. PDG-DUP allows the user limited control over this behavior by means of the `COMMON` variable, but, to our knowledge, Bauhaus' ccdiml and CCFinder have no such controls.

7 Conclusions

7.1 Contributions

First, our results confirm the belief that some crosscutting concerns are implemented by similar pieces of code, which are scattered throughout a system. Our case study shows that these pieces of code can contribute up to 25 percent to the code size. Large gains in terms of maintainability and evolvability are thus to be expected from methods supporting the identification and refactoring of these crosscutting concerns.

Second, we have evaluated to what extent the code of five crosscutting concerns is identified by three clone detection techniques. To that end, we manually annotated the code of five specific concerns in an industrial C application and analyzed the recall, precision, and average precision obtained by clone classes yielded by the three clone detection tools. It turns out that the clone classes obtained by Bauhaus' ccdiml can provide the best match with the Range Checking, NULL-value Checking, and Error Handling concerns. However, CCFinder's clone classes perform almost equally well for NULL-value Checking and Error Handling. The remaining concerns, i.e., Tracing and Memory Error Handling, can best be matched by clone classes of PDG-DUP.

Finally, we discussed how the results obtained in the case study pose an upper limit to the suitability of using the studied clone detectors for aspect mining purposes. In particular, since Error Handling code is matched badly by all three clone detectors, automatic aspect mining approaches using (one of) these clone detectors cannot be expected to adequately find code belonging to the Error Handling concern. On the other end of the spectrum, code of the NULL-value Checking concern could be found very well by using CCFinder or Bauhaus' ccdiml.

7.2 Future Work

The crosscutting concerns we considered in the case study also occur in a range of other ASML components. We will investigate how we can identify the code belonging to these concerns without manual annotations, using the clone classes found in the *CC* component as a starting point. In other words, the *CC* clone classes can be used as seeds for the crosscutting concern identification in other components of the ASML code base.

The code base studied in this paper is confidential, which hinders exact reproduction of the experiment by other researchers. However, we believe that crosscutting concerns similar to the ones studied here are also present in publicly accessible source code bases. For instance, NULL-value checking, Error Handling, and Tracing are common concerns for any reasonably large C system. Therefore, it would be worthwhile and feasible to reproduce the experiment on a publicly accessible source base.

Clone classes can be characterized by simple metrics like the number of clones contained in a clone class, or the number of lines covered by the class, but more complex metrics can be derived as well, such as the distribution of clones over different files. Metrics such as these could be used to study the nature of clone classes that capture crosscutting concerns (and those that do not), given that these clone classes are known. The case study presented in this paper shows one way of identifying such clone classes, i.e., using manually obtained annotations. Consequently, relationships between clone class metrics and recall and precisions levels could be discovered based on the results of our case study. Such relations could then be tested on other systems, including those written in other languages.

Aspect mining techniques based on clone detection can possibly benefit from knowing which clone class characteristics relate to crosscutting concerns. For instance, in [37], we discuss how a number of clone class metrics can be used to filter clone detection results.

Finally, we are working toward the elimination of the NULL-value Checking and Tracing concerns from the original source code [38]. The implementations of these concerns are replaced by domain-specific solutions, which subsequently generate aspect-oriented code. An interesting issue with respect to clone detection is the suitability of the detected clones for such (semiautomatic) refactorings. For instance, clones that do not consist of complete syntactical units are likely unsuitable for this purpose. Furthermore, clones that have context dependencies (data or control) require additional effort to be successfully extracted. The PDG-DUP clone detector appears to be most suitable for such refactoring activities since it both respects syntactic integrity and includes context dependencies in its clones.

Acknowledgments

Partial support was received from SENTER (CWI, project IDEALS, hosted by the Embedded Systems Institute). Arie van Deursen was also supported by ITEA (Delft University of Technology, project MOOSE, ITEA 01002). The authors would like to thank Jan Heering and various anonymous reviewers for thoroughly reading drafts of this paper and offering many suggestions for improvement. They would also like to thank Rainer Koschke, Toshihiro Kamiya, and Raghavan Komondoor for the availability of their respective tools, Bauhaus, CCFinder, and PDG-DUP.

References

[1] P. Tarr, H. Ossher, W. Harrison, and S.M.J. Sutton, "N Degrees of Separation: Multi-Dimensional Separation of Concerns," *Proc. 21st Int'l Conf. Software Eng. (ICSE '99),* pp. 107-119, May 1999.

[2] E. Soloway, R. Lampert, S. Letovsky, D. Littman, and J. Pinto, "Designing Documentation to Compensate for Delocalized Plans," *Comm. ACM,* vol. 31, no. 11, pp. 1259-1267, 1988.

[3] J. Hannemann and G. Kiczales, "Overcoming the Prevalent Decomposition in Legacy Code," *Proc. ICSE Workshop Advanced Separation of Concerns,* May 2001, http://www.cs.ubc.ca/~jan/amt.

[4] M. Kim, L. Bergman, T.A. Lau, and D. Notkin, "An Ethnographic Study of Copy and Paste Programming Practices in OOPL," *Proc. Int'l Symp. Empirical Software Eng. (ISESE '04),* pp. 83-92, Aug. 2004.

[5] J. Johnson, "Identifying Redundancy in Source Code Using Fingerprints," *Proc. IBM Centre for Advanced Studies Conf. (CASCON '93),* pp. 171-183, Oct. 1993.

[6] S. Ducasse, M. Rieger, and S. Demeyer, "A Language Independent Approach for Detecting Duplicated Code," *Proc. Int'l Conf. Software Maintenance (ICSM '99),* pp. 109-118, 1999.

[7] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A Multi-linguistic Token-Based Code Clone Detection System for Large Scale Source Code," *IEEE Trans. Software Eng.,* vol. 28, no. 7, pp. 645-670, July 2002.

[8] B.S. Baker, "On Finding Duplication and Near-Duplication in Large Software Systems," *Proc. Second Working Conf. Reverse Eng. (WCRE '95),* pp. 86-95, July 1995.

[9] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone Detection Using Abstract Syntax Trees," *Proc. Int'l Conf. Software Maintenance (ICSM '98),* pp. 368-377, Nov. 1998.

[10] R. Komondoor and S. Horwitz, "Using Slicing to Identify Duplication in Source Code," *Proc. Eighth Int'l Symp. Static Analysis (SAS '01),* pp. 40-56, July 2001.

[11] J. Krinke, "Identifying Similar Code With Program Dependence Graphs," *Proc. Eight Working Conf. Reverse Eng. (WCRE '01),* pp. 301-109, 2001.

[12] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics," *Proc. Int'l Conf. Software Maintenance (ICSM '96),* pp. 244-254, Nov. 1996.

[13] A. Marcus and J.I. Maletic, "Identification of High-Level Concept Clones in Source Code. " *Proc. 16th IEEE Int'l Conf. Automated Software Eng. (ASE '01),* pp. 107-114, Nov. 2001.

[14] G. Mishne and M. de Rijke, "Source Code Retrieval Using Conceptual Similarity," *Proc. 2004 Conf. Computer Assisted Information Retrieval (RIAO '04),* pp. 539-554, Apr. 2004.

[15] M. Rieger, S. Ducasse, and G. Golomingi, "Tool Support for Refactoring Duplicated OO Code," *Proc. European Conf. Object-Oriented Programming (ECOOP '99),* pp. 177-178, June 1999.

[16] M. Balazinska, E. Merlo, M. Dagenais, B. Laguë, and K. Kontogiannis, "Advanced Clone-Analysis to Support Object-Oriented System Refactoring," *Proc. Seventh Working Conf. Reverse Eng. (WCRE '00),* pp. 98-107, Nov. 2000.

[17] F. van Rysselberghe and S. Demeyer, "Evaluating Clone Detection Techniques from a Refactoring Perspective," *Proc. Ninth IEEE Int'l Conf. Automated Software Eng. (ASE '04),* pp. 336-339, Sept. 2004.

[18] R. Fanta and R. Václav, "Removing Clones from the Code," *J. Software Maintenance: Research and Practice,* vol. 11, no. 4, pp. 223-243, July/Aug. 1999.

[19] F. van Rysselberghe and S. Demeyer, "Reconstruction of Successful Software Evolution Using Clone Detection," *Proc. Sixth Int'l Workshop Principles of Software Evolution (IWPSE '03),* pp. 126-130, Sept. 2003.

[20] A. Walenstein, "Problems Creating Task-Relevant Clone Detection Reference Data," *Proc. 10th Working Conf. Reverse Eng. (WCRE '03),* pp. 285-294, Nov. 2003.

[21] A. Walenstein and A. Lakhotia, "Clone Detector Evaluation Can Be Improved: Ideas from Information Retrieval," *Proc. Second Int'l Workshop the Detection of Software Clones (IWDSC '03),* pp. 11-12, Nov. 2003.

[22] W.G. Griswold, J.J. Yuan, and Y. Kato, "Exploiting the Map Metaphor in a Tool for Software Evolution," *Proc. Int'l Conf. Software Eng. (ICSE '01),* pp. 265-274, Mar. 2001.

[23] G.C. Murphy, W.G. Griswold, M.P. Robillard, J. Hannemann, and W. Leong, "Design Recommendations for Concern Elaboration Tools," *Aspect-Oriented Software Development,* R.E. Filman, et al., eds., pp. 507-530, 2005.

[24] C. Zhang and H.-A. Jacobsen, "PRISM is Research In aSpect Mining," *OOPSLA Companion,* J.M. Vlissides and D.C. Schmidt, eds., pp. 20-21, 2004.

[25] C. Zhang and H.A. Jacobsen, "Quantifying Aspects in Middleware Platforms," *Proc. Second Int'l Conf. Aspect-Oriented Software Development (AOSD '03),* pp. 130-139, Mar. 2003.

[26] R. Ettinger and M. Verbaere, "Untangling: A Slice Extraction Refactoring," *Proc. Third Int'l Conf. Aspect-Oriented Software Development (AOSD '04),* pp. 93-101, Mar. 2004.

[27] S. Breu and J. Krinke, "Aspect Mining Using Event Traces," *Proc. 19th IEEE Int'l Conf. Automated Software Eng. (ASE '04),* pp. 310-315, Sept. 2004.

[28] D. Shepherd, E. Gibson, and L.L. Pollock, "Design and Evaluation of an Automated Aspect Mining Tool," *Proc. Int'l Conf. Software Eng. Research and Practice (SERP '04),* pp. 601-607, June 2004.

[29] T. Tourwé and K. Mens, "Mining Aspectual Views Using Formal Concept Analysis," *Proc. Fourth Int'l Workshop Source Code Analysis and Manipulation (SCAM '04),* pp. 97-106, Sept. 2004.

[30] M. Marin, A. van Deursen, and L. Moonen, "Identifying Aspects Using Fan-In Analysis," *Proc. 11th Working Conf. Reverse Eng. (WCRE '04),* pp. 132-141, Nov. 2004.

[31] M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonella, and T. Tourwé, "A Qualitative Analysis of Three Aspect Mining Techniques," *Proc. Int'l Workshop Program Comprehension (IWPC '05),* May 2005.

[32] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn, "Using Aspect C to Improve the Modularity of Path-Specific Customization in Operating System Code," *Proc. Joint European Software Eng. Conf. (ESEC '01),* pp. 88-98, June 2001.

[33] C. van Rijsbergen, *Information Retrieval,* second ed. London: Butterworths, 1979.

[34] "Project Bauhaus," http://www.bauhaus-stuttgart.de, 2005.

[35] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwé, "An Evaluation of Clone Detection Techniques for Identifying Crosscutting Concerns," *Proc. IEEE Int'l Conf. Software Maintenance (ICSM '04),* pp. 200-209, Sept. 2004.

[36] R. Holt, "Structural Manipulations of Software Architecture Using Tarski Relational Algebra," *Proc. Fifth Working Conf. Reverse Eng. (WCRE '98),* pp. 210-219, Oct. 1998.

[37] M. Bruntink, "Aspect Mining Using Clone Class Metrics," CWI Technical Report SEN-E0502, Centrum voor Wiskunde en Informatica (Center for Math. and Computer Science), Amsterdam, Netherlands, Feb. 2005.

[38] M. Bruntink, A. van Deursen, and T. Tourwé, "Isolating Idiomatic Crosscutting Concerns," *Proc. IEEE Int'l Conf. Software Maintenance (ICSM '05),* Sept. 2005.

Magiel Bruntink received the MSc degree in computer science from the University of Amsterdam in 2003. Starting in October 2003, he has been working as a graduate student at the Centrum voor Wiskunde en Informatica in Amsterdam. His research interests include reverse engineering, program analysis, and programming languages. Most of his current work is done in the scope of the Ideals project, which consists of a collaboration between academic and industrial partners. He is a student member of the IEEE Computer Society.

Arie van Deursen received the MSc degree (1990) from the Vrije Universiteit Amsterdam and the PhD degree (1994) from the University of Amsterdam. He is a professor of software engineering at Delft University of Technology and research leader at CWI, the Dutch National Research Institute in Mathematics and Computer Science. His research interests include aspect-oriented software development, program analysis, software testing, and program comprehension. He is a member of the IEEE, the IEEE Computer Society, the ACM, and the ACM Special Interest Group on Software Engineering. He was program chair of the Working Conference on Reverse Engineering in 2002 and 2003.

Remco van Engelen received the MSc degree in computer science from Eindhoven University of Technology in 1996. Since September 1996, he has been working at ASM Lithography, a leading supplier of semiconductor manufacturing equipment based in the Netherlands. Starting in 2002, he has participated in the Ideals project. His research interests include program analysis, program transformation, and practical software engineering.

Tom Tourwé received the degree of Licentiate in computer science in 1997 and the PhD degree in science in 2002 at the Vrije Universiteit Brussel. He is currently associated with the Centrum voor Wiskunde en Informatica, based in Amsterdam, The Netherlands, where he works as a postdoctoral researcher in the Ideals project. His main research interests lie in the broad area of software evolution and include aspect-oriented evolution and reengineering in particular. He published several peer-reviewed articles on these topics in international journals, conferences, and workshops, and organized several workshops on those themes. He is a member of the IEEE and the IEEE Computer Society.