An Evaluation of Clone Detection Techniques for Identifying Crosscutting Concerns

Magiel Bruntink, Arie van Deursen,* Tom Tourwé
Centrum voor Wiskunde en Informatica
P.O. Box 94079, 1090 GB Amsterdam
The Netherlands
{Magiel.Bruntink,Arie.van.Deursen,Tom.Tourwe}@cwi.nl

Remco van Engelen
ASML Netherlands B.V.
De Run 6501, 5504 DR Veldhoven
The Netherlands
Remco.van.Engelen@asml.com

Abstract

Code implementing a crosscutting concern is often spread over many different parts of an application. Identifying such code automatically greatly improves both the maintainability and the evolvability of the application. 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 a developer to refactor the code, so that it uses modern and more advanced abstraction mechanisms, thereby restoring 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 four specific concerns in an industrial C application, and analyze to what extent clone detection is capable of finding these concerns. We consider our results as a stepping stone toward an automated "concern miner" based on clone detection.

1. Introduction

The tyranny of the dominant decomposition [17] implies that no matter how well a software system is decomposed into modular units, some functionality (often called a *concern*) will always crosscut that decomposition. In other words, such functionality cannot be cleanly captured 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 localize 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 localizing crosscutting code is a time-consuming and error-prone activity.

Aspect-oriented software development (AOSD) has been proposed for solving the problem of the dominant decomposition. Aspect-oriented programming languages add an abstraction mechanism (called an *aspect*) to existing (object-oriented) programming languages. 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 should be evolved into aspect-oriented applications. Once again, this requires identifying the crosscutting code within all modules of the application.

In order to support developers in evolving their applications, and improving the maintainability and understandability of their code, some form of automated support for identifying crosscutting code is indispensable. In this paper, we report upon a first experiment in which we assess the effectiveness of clone detection techniques for that particular purpose. Clone detection techniques are promising in this respect, because by definition crosscutting code is not well modularized and can thus not be easily reused. Consequently, such code is typically duplicated over the entire application, and could be identified using advanced clone detection algorithms.

The experiment addresses crosscutting concerns in an industrial software component written in C and in use at ASML. A domain expert manually marked occurrences of four types of crosscutting concerns (error handling, tracing, parameter checking, and memory management). We then applied two different clone detection techniques in order to see how much of the code belonging to these concerns can be found in an automatic way.

This paper is structured as follows. In the next section, we discuss the problem statement in more detail, followed by a short overview of clone detection techniques in Section 3. Then, in Section 4, we describe the case study and

*CWI and Delft University of Technology

the four different concerns of interest in it. Subsequently, in Section 5 we detail the approach used to evaluate the capability of clone detection to find these concerns. In Section 6 we cover and explain the results obtained. We conclude this paper with a discussion of related work, future work, and a summary of the paper's main contributions.

2. Problem Statement

Localizing a crosscutting concern manually within the source code of a large application typically involves the following three steps:

1. To identify a particular crosscutting concern, a developer should already have some idea about how it appears in the source code. In other words, a developer should identify a particular pattern that is used to implement the concern. Clearly, this already requires a good understanding of the concern itself, the source code and the way it implements the concern. Such a *concern pattern* is often referred to as a *seed*;

2. Assuming a developer identified a pattern associated to the concern, he should browse and inspect the source code of the application in order to look for that pattern. Especially in large-scale applications involving millions of lines of code, this is quite a daunting task. We can easily imagine a developer overlooking some code that implements the concern, or mistakenly identifying code as if it implements the concern;

3. Due to the crosscutting nature of the concern, more often than not, the source code implementing the concern will be intertwined with code implementing other concerns. A developer interested in only one concern should thus be able to discriminate the different concerns implemented by the code. Again, this requires a good understanding of the concerns and of the source code.

Clearly, identifying crosscutting concern code is thus an error-prone and time-consuming task, and each of the three steps above could benefit greatly from some form of automated support.

In order to provide such automated support, an intrinsic property of crosscutting concern code can be exploited. Typically, source code implementing a crosscutting concern involves a great deal of duplication. First of all, since such code cannot be captured cleanly inside a single abstraction, it cannot be reused. Therefore, developers are forced to write the same code over and over again, and are tempted to just copy and paste the code and adapt it slightly to the context. Alternatively, they may use particular coding conventions and idioms, which also exhibit similar code. Second, crosscutting concerns often involve 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 all these observations that clone detection techniques might be ideal candidates for identifying crosscutting concern code, since they automatically detect duplicated code in an application's source code. Moreover, because of the significant research effort spent on clone detection, the techniques and algorithms available are both stable and scalable.

In this paper we seek to evaluate this hypothesis. In particular, we want to assess the usefulness and accuracy of clone detection techniques for finding crosscutting code. We will manually identify some prevalent concerns and evaluate how much of the code that implements these concerns can be found by the clone detection algorithms. Our increased understanding of how well clone detection captures crosscutting code will then be an important first step toward a "concern miner" that can offer automated support for all three steps involved in the localization of crosscutting concerns.

3. Clone Detection

Clone detection techniques attempt at finding duplicated code, which may have undergone minor changes afterward. The typical motivation for clone detection is to factor out copy-paste-adapt code, and replace it by a single procedure.

Several clone detection techniques have been described and implemented:

Text-based techniques [10, 5] 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 [11, 1] apply a lexical analysis (tokenization) to the source code, and subsequently use the tokens as a basis for clone detection.

AST-based techniques [2] 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 [12, 13] go one step further in obtaining a source code representation of high abstraction. Program dependence graphs (PDGs) contain information of semantical nature, such as control- and data flow of the program.

Metrics-based techniques [14] are related to hashing algorithms. For each fragment of a program the values of a number of metrics is calculated, which are subsequently used to find similar fragments.

Following Walenstein [18], clone detection adequacy depends on application and purpose. Finding crosscutting con-

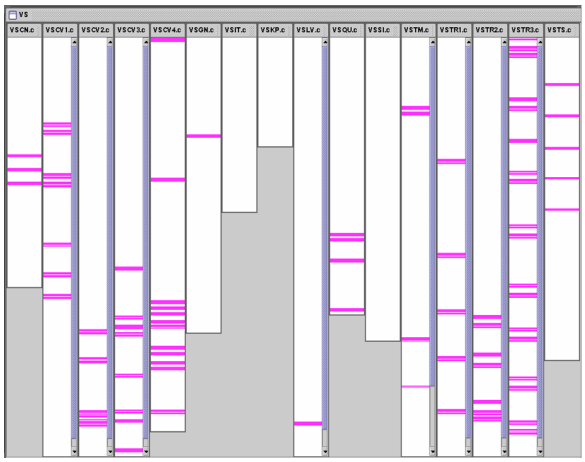


Figure 1. Distribution of the memory error handling concern over 19,000 lines of code

cerns is a completely new application area, potentially requiring specialized types of clone detection.

4. Case Study

4.1. Background

Our paper is based on a small experiment on a software component (called *VS*) of 19,000 lines of C code, part of the larger code base (comprising over 10 million lines of code) of ASML, the world market leader in lithography systems based in Veldhoven, The Netherlands. The VS component is responsible for the conversion of data between several data structures and other utilities used by communicating components.

Developers working on this component express 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 parameter checking (explained below).

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 these crosscutting concerns in all applicable modules, similar pieces of code are scattered throughout the system. Clearly, large benefits in code size, quality and comprehensibility are to be expected if such concerns could be handled in a more systematic and controlled way.

4.2. Considered Concerns

A domain expert manually marked places in the VS component dealing with four different crosscutting concerns. Each

Concern	LOC	Fraction
Error handling	1716	9%
Dynamic execution tracing	1539	8%
Function parameter checking	1441	7%
Memory allocation handling	1110	6%
Total	5806	31%

Table 1. Code percentages devoted to various concerns, in a 19 KLOC component.

line in the application was annotated with at most one mark, and as a result, each line belongs to at most one of the concerns described below, or to no concern.

- General error handling and administration; this code 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.
- Dynamic execution tracing; logging the values of input and output parameters of C functions to facilitate debugging.
- Function pre and post condition checking; covering pointer and array bound checks.
- Dedicated handling of errors originating from C memory management.

The concern markers provided by the domain expert not only allow us to obtain an intuition about the amount of code devoted to crosscutting concerns, but also enables us to subsequently apply clone detection techniques in order to get as close as possible to retrieving these crosscutting concerns from the code. Note that this approach does not automate step 1 of the process outlined in Section 2. Although clone detection techniques do not need a seed to perform their analysis, and could thus be used to uncover the pattern of a concern as well, we did not consider that option yet. Step 2 of the process is automated, however, since the clone detector inspects the source code automatically.

All together, these concerns comprise roughly 31% of the code. The details are shown in Table 1, while Figure 1 illustrates the scattered nature of these concerns by displaying the distribution of the code fragments belonging to the memory error handling concern. The vertical bars represent the files of the 19 KLOC component, and within each vertical bar, horizontal lines of pixels correspond to lines of source code within the file. Coloured lines are part of the memory error handling concern. The other concerns exhibit a similarly scattered distribution.

5. Approach Taken

5.1. Clone Detectors

The first step in our experiment consists of detecting clones in the selected component. For this purpose we have used two different tools, which implement different clone detection techniques. First, we used the clone detection tool contained in Bauhaus[1] (version 4.7.2), a toolset aimed at supporting program understanding and reengineering. The clone detector, called 'ccdiml', is an implementation of a variation of Baxter's approach to clone detection[2], and thus falls in the category of AST-based clone detectors. Second, we used CCFinder[11] (version 7.1.1), a clone detection tool based on tokenized representations of source code.

We have selected these specific clone detection tools for several reasons. First, both tools offer excellent performance in terms of running time and memory usage. Second, we felt that the code pieces implementing our selected concerns are similar mostly at the syntactical level, hence the choice for an AST-based clone detector, i.e. Bauhaus' ccdiml. Third, we were interested in the gain of precision and recall obtained by using an AST-based clone detector compared to those using more textually-oriented approaches.

Both detection tools are highly configurable with respect to the types of clones they detect. 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 7 tokens long. For Bauhaus' ccdiml, a minimum length of 2 lines was selected. These values were chosen such that we would not expect that pieces of concern code were missed because of the minimum clone length. Furthermore, in the case of CCFinder, a smaller minimum clone length resulted in an intractable number of clones. Bauhaus' ccdiml was also configured to compare all statements of the source code, as opposed to comparing only functions. CCFinder did not require such a setting. 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.

5.2. Clone Classes

Most 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* [11].

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 indeed describes an equivalence relation between code fragments [11], 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 our experiment, i.e. either exact or parametrized clones, we feel justified in augmenting 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 et. al.[9].

5.3. Concern Coverage

Up to now, we have described how we obtained two sources of information. On the one hand, we have manually annotated source code lines of a 19 KLOC component. Each line of code of the component is thus known to be either error handling, memory error handling, tracing or parameter checking code, or other code. On the other hand, we have obtained two sets of clone classes by means of two clone detectors, Bauhaus' ccdiml and CCFinder. These two sets of clone classes are the subjects of our further analysis.

A clone class defines a (non-contiguous) region of source code that is related according to some clone detector. The manually annotated source code is also partitioned in several (non-contiguous) regions, namely those lines of source code that implement error handling, memory error handling, tracing, parameter checking code, and other code. With regard to our goal, an interesting criterion for evaluation is the extent to which the region defined by a clone class matches the region defined by a concern. In fact, we aim at evaluating the usefulness of a clone detector for identifying concern code by analyzing the extent to which each of the concerns we consider is 'covered' by the set of clone classes yielded by the clone detector. The number of clone classes needed to cover one particular concern completely (in part) indicates how well the clone detector is able to identify the concern.

[1] URL: `http://www.bauhaus-stuttgart.de/`

In order to perform this evaluation, we use grok to process the sets of clone classes of both clone detectors separately. For each of the concerns we consider, we try to find an ordered selection of clone classes that does a good job at 'covering' the region of code defined by the concern in question. A source code line of a concern is covered by a clone class if it is included in one of the clones (code fragments) of the clone class.

For each concern, we then proceed as follows: for all of the clone classes in the set, we calculate which concern lines are covered by each clone class. The clone class that covers the most lines of the concern is selected, and the concern lines that are covered will no longer be considered during the remainder of the algorithm. Subsequently, the algorithm will select the clone class that covers the most of the remaining concern lines, and so on until no more concern lines are covered by any clone class. If it occurs that multiple clone classes cover an equal number of concern lines, we select the clone class that contains the least number of non-concern lines. Similar to lines belonging to a concern, non-concerns lines are also considered at most once.

6. Obtained Results

Our primary goal is finding the code belonging to a certain concern. Therefore, in our algorithm to select the clone classes (see Section 5), we favor coverage and sacrifice precision (defined below). Arguably, other goals require different criteria to rank the clone classes. For example, in order to identify opportunities for (automatic) refactoring, precision would be the primary issue. We plan to explore these possibilities in the future.

In order to evaluate to what extent the clone detectors meet our goal, we investigate the level of *concern coverage* met by the clone classes. *Concern coverage* is the fraction of a concern's source code lines that are covered by the first n selected clone classes. Using the selection algorithm described in Section 5 we obtain the results displayed in Figure 2(a) and Figure 2(b) for Bauhaus' ccdiml and CCFinder, respectively.

Additionally, we evaluate the *precision* obtained by the first n selected clone classes. *Precision* is defined as follows:

$$\text{precision}(n) = \frac{\text{concernLines}(n)}{\text{totalLines}(n)},$$

where n indicates the first n selected clone classes, concernLines equals the number of concern code lines covered by the first n selected clone classes, and likewise totalLines equals the total number of lines covered by the first n selected clone classes. Figure 2(c) and Figure 2(d) show the precision obtained by the first n selected clone classes for Bauhaus' ccdiml and CCFinder, respectively.

Observe that as the number of clone classes considered increases, the coverage displays a monotonic growth, whereas the precision tends to decrease. The highest coverage is less than 100% in all cases: the remaining percentage corresponds to concern code that is coded in such a unique way that it does not occur in any clone class. For example, Figure 2(a) and Figure 2(b) show that 5% of the memory error handling code is not part of any clone class.

We are primarily interested in achieving sufficient coverage without loosing too much precision. Therefore, we will focus on the number of clone classes needed to cover most of a concern, where we will consider 80% to be a sufficient coverage level.

6.1. Memory Error Handling

Using 9 clone classes is enough to sufficiently cover the memory error handling concern for both Bauhaus' ccdiml and CCFinder, resulting in 69% and 52% precision, respectively.

We observe that CCFinder yields a clone class that already covers 45% of the concern code. This particular clone class contains 96 clones which are 6 lines in length. Figure 3 shows an example clone from this class. While the lines marked with 'M' belong to the memory handling concern, only the lines marked with 'C' are included in the clones. CCFinder allows clones to start and end with little regard to syntactic units. In contrast, Bauhaus' ccdiml does not allow this, due to its AST-based clone detection algorithm.

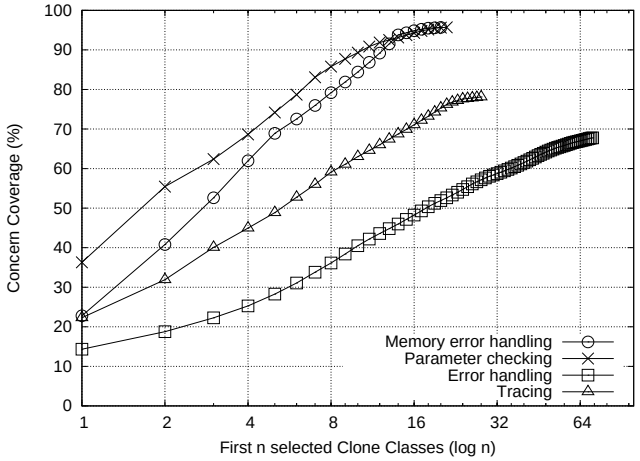
```
M C if (r != OK)
M C {
M C   ERXA_LOG(r, 0, ("PLXAmem_malloc failure."));
M C
M C   ERXA_LOG(VSXA_MEMORY_ERR, r,
M C           ("%s: failed to allocated %d bytes.",
M             func_name, toread));
M
M     r = VSXA_MEMORY_ERR;
M   }
```

Figure 3. CCFinder clone covering memory error handling.

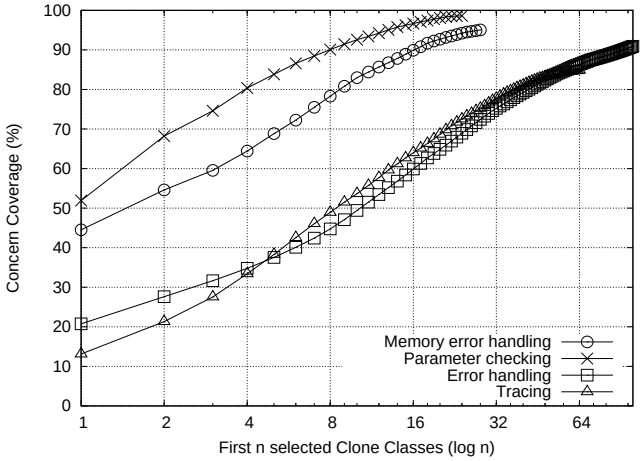
Furthermore this clone class does not cover memory error handling code exclusively. In Figure 2(d), note that the precision obtained for the first clone class is roughly 82%. Through inspection of the code we found that some of the clones do not cover memory error handling code at all, but code that is similar at the syntactical level, yet semantically different.

6.2. Parameter Checking

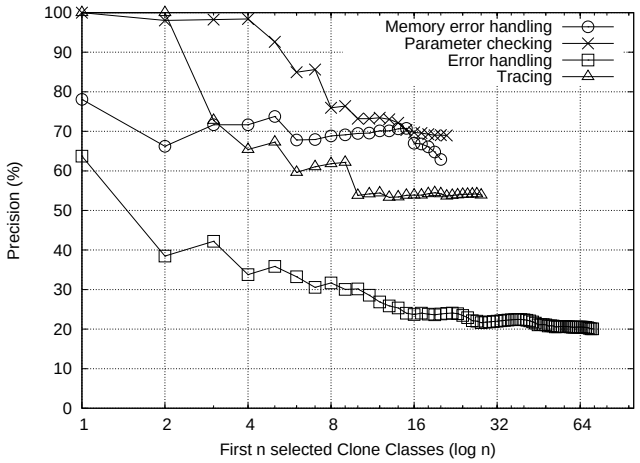
Our results show that the parameter checking concern is found very well by both clone detectors: using 7 clone classes of Bauhaus' ccdiml is sufficient to cover 80% of the concern, while for CCFinder we can suffice with 4 clone



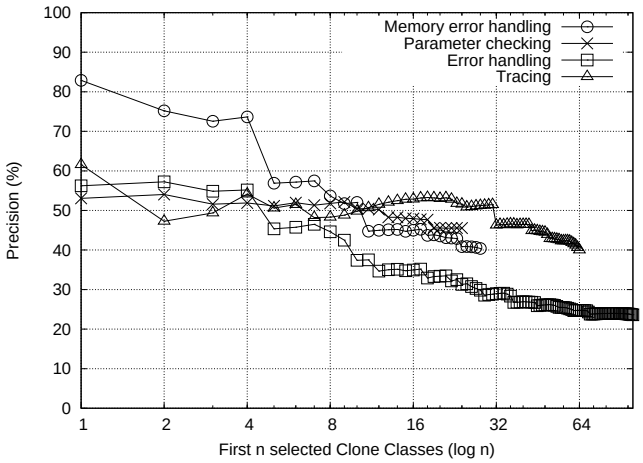
(a) Concern coverage for Bauhaus' ccdiml.



(b) Concern coverage for CCFinder.



(c) Precision for Bauhaus' ccdiml.



(d) Precision for CCFinder.

Figure 2. Results of the clone detection experiment.

classes. Precision obtained using the selected clone classes is 85% Bauhaus' ccdiml and 52% for CCFinder.

The clone class that was selected first in the case of Bauhaus' ccdiml captures roughly 40% of the concern, at a precision of 100%. This class consists of 79 clones, spanning 523 lines of parameter checking code. In Figure 4 we show an example clone of this clone class. The lines marked with 'P' belong to the parameter checking concern and those marked with 'C' are the lines included in the clone. Inspection of the example shows that clones in this class correspond to the pieces of code that handle the case of invalid parameter values, while the code doing the actual checking of the parameters is not covered. The pieces of checking code can differ strongly due to a varying number of parameters, or varying types of the parameters.

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

Figure 4. Bauhaus' ccdiml clone covering parameter checking.

The second clone class for Bauhaus' ccdiml contains clones similar to the one in Figure 4, but including the actual checking code. This clone class contains 48 clones, spanning 276 lines of parameter checking code.

6.3. Error Handling

Of the concerns we consider, error handling is clearly the worst in terms of both coverage and precision. All the clone classes yielded by Bauhaus' ccdiml together do not succeed in covering 80% of the concern, while for CCFinder we reach sufficient coverage only after selecting 42 clone classes. At that point precision is well below 50%.

The error handling concern can be partitioned into three sub-concerns: *initialization*, *error linking* and *skipping*. First, the *initialization* sub-concern 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 experiments have shown that, on the one hand, the initialization and error linking sub-concerns are generally found well, i.e. coverage exceeding 90% while using a very limited number of clone classes. On the other hand, the skipping sub-concern is found very badly, which explains why the error handling concern in general is found badly.

Consider the code fragment in Figure 5, a simple example of code belonging to the skipping sub-concern. The lines marked with 'S' are those belonging to the skipping concern. The skipping sub-concern accounts for 1231 lines of code, i.e. 6,5% of the VS 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.

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

Figure 5. Instance of the skipping concern.

The clone classes yielded by both clone detectors do not provide good coverage of 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 5. Furthermore, the code that appears inside the if statements can differ greatly. As a result no clones classes are found that cover just the skipping concern. However, some clone classes cover the skipping sub-concern by accident, i.e. the clones cover a large number of non-concern lines compared to the number of skipping lines covered.

6.4. Tracing

Compared to the other concerns, the coverage obtained for tracing is mediocre: the clone classes of Bauhaus' ccdiml reach 78% coverage all together, while we reach sufficient coverage using 37 clone classes of CCFinder. Precision obtained by the clone classes of Bauhaus' ccdiml is higher than for the clone classes of CCFinder, especially for the first 9 selected clone classes. For both clone detectors, precision nears 50% as coverage of the concern approaches 80%.

The first two clone classes for Bauhaus' ccdiml together cover 32% of the tracing code, while maintaining 100% precision. An example clone from the first clone class is shown in Figure 6. In total, 71 clones of this class are present in the VS 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.

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

Figure 6. Bauhaus' ccdiml clone covering tracing.

All the code belonging to the tracing concern is very similar to the example in Figure 6. 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 result, we also find clone classes which consist of calls to `THXAtrace(...)` with 5 arguments, 6 arguments, and so on. In fact, the second clone class selected for Bauhaus' ccdiml contains clones of the `THXAtrace(...)` function call with 6 arguments.

CCFinder does not yield clones of the `THXAtrace(...)` function call with less than 6 arguments, simply because we have limited the minimum size of a clone to 7 tokens (see Section 5). However, clone classes including `THXAtrace(...)` function calls with less than 6 arguments do turn up, but they also include a number of non-tracing lines. A clone belonging to the first clone class selected for CCFinder is shown in Figure 7. 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 Figure 7, 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
  C return r;
  C }
```

Figure 7. CCFinder clone covering tracing.

6.5. Discussion

For both Bauhaus' ccdiml and CCFinder, we have seen that the parameter checking and memory error handling concerns are covered sufficiently well (80%) using a limited number of clone classes. Furthermore, the level of precision obtained by the clone classes of Bauhaus' ccdiml in the parameter checking case is very high: the first 4 selected clone classes obtain a precision as high as 98%.

In contrast, the tracing and error handling concerns do not show the same results. While the clone classes yielded by CCFinder do reach a coverage of 80%, the number of clone classes required is quite high. Both of these concerns are not covered sufficiently well by the clone classes of Bauhaus' ccdiml, although the coverage of tracing is almost sufficient (78%). Additionally, the obtained precision is low. An exception is the precision obtained for the tracing concern by the first 2 clone classes of Bauhaus' ccdiml: they cover 32% of the tracing code at 100% precision. However, if more clone classes are considered the precision degrades quickly.

We observe that the precision obtained by the clone classes of Bauhaus' ccdiml is generally higher than for the clone classes of CCFinder, though there are a few exceptions. Inspection of the results has shown that clones yielded by CCFinder tend to begin and end with little regard to the syntactic structure of the source code. An example can be seen in Figure 3. Due to the AST-based technique, clones yielded by Bauhaus' ccdiml are constrained to (sequences of) syntactic units. Both clone detectors attempt to find clones of maximal length, which entails that they try to extend clones as far as possible. Because CCFinder is not limited to syntactic units, but instead uses (generally smaller) tokens, clones are also more likely to be extended. Consequently, clones covering instances of a concern are more likely to also cover a number of non-concern lines which (coincidentally or not) often occur in the presence of a concern instance.

However, we also observe an advantage of the token-based approach. Figure 3 shows an instance of the memory error handling concern, being covered in part by a CCFinder clone. This particular clone class alone covers 45% of the concern (see Figure 2(b)). A similar clone class is found for parameter checking, yet resulting in much lower precision. Clone classes like these, covering similar parts of concern instances, are less likely to be found by Bauhaus' ccdiml due to the restriction of clones to (sequences of) syntactic units. However, considering that ultimately our goal is to refactor the identified concerns into aspects, it is clear that it is diserable to have syntactically sound units representing the concern.

We conclude this discussion by considering the implications of the results for our future work. First, evaluating the clone classes in terms of both concern coverage and precision identified many clone classes that will be of value during our research within the context of the ASML code base. The component we considered in this paper is a small, but representative, example of the components that exist within this code base. Our larger goal is to identify crosscutting concerns in all these components, and support the refactoring of these concerns into a more modular solution.

Second, in this paper we showed that for some concerns, clone detection techniques can be used to obtain a sufficiently complete coverage of those concerns. These results will allow us to move ahead with research, testing the general applicability of clone detection techniques to the aspect mining problem. The next step would be to evaluate the use of clone detection techniques to identify concern code fully automatically. Our results allow us to derive characterizations of clone classes that provide high coverage of the concerns, which could subsequently be used to filter clone detection results of other systems. In Section 8 we describe how we plan to use the results obtained in this paper.

Finally, clone detection techniques work because the code implementing the concerns has been developed using a disciplined and idiomatic approach. In other words, the development process provides strict rules on the implementation of the concerns in question, resulting in many similar pieces of code. We view the idiomatic nature of the development process at ASML as a major condition to the applicability of clone detection techniques for the purpose of aspect mining, and thus our results. The language used to implement the concerns appears to be of lesser importance, although we do not yet have results to support that claim. Obviously, the type of clone detection applied is of major importance as well.

7. Related Work

The considerable research effort spent on clone detection techniques makes them both very stable and scalable to large-scale programs. In recognition of this fact, aspect mining is only one particular application of using clone detection techniques for reasoning about software. [15] for example, uses a clone detection algorithm to study the evolution of a software system. In particular, they try to distinguish *move method* refactorings that were applied when evolving one version of the software into another.

Although 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. Most notably, [4] reports upon an experiment using aspect-oriented techniques to modularize the implementation of prefetching within page fault handling in the FreeBSD operating system kernel. To this extent, 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 instead of automatically.

Although research on aspect mining is still in its infancy, several prototype tools have already been developed that support developers in identifying crosscutting code. Many of these tools are semi automatic, which means they require some form of input by the developer. Those are discussed in the next paragraph. More advanced tools, that are able to identify aspects without human intervention, are appearing as well, and will be discussed afterward.

The *Aspect Browser* [7] is a programming environment that provides *text-based mining*, which means it relies on 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. The *Aspect Mining Tool* [8] 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 [19] 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 sign 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. [6] discusses 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. 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 proposed in literature as well. [3] proposes a tool that dynamically analyzes Java software to identify aspects. To this extent, 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 a method `b`, or a call to a particular method `c` that always occurs inside a call to a method `d`. Such patterns could point to before/after aspects. [16] presents a tool that uses a clone detection algorithm based on the program dependence graph. The tool identifies possible aspects fully automatically, focusing currently on before advice. Although the tool still consumes considerable resources, the initial results are promising.

8. Future Work

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

Clone classes can be characterized by simple measures like the number of clones contained in such a class, or the number of lines covered by the class, but more complex measures can be derived as well, such as the distribution of clones over different files. Such measures could be useful to filter clone detection results for the purpose of automatic aspect mining. Based on the results described in this paper, we could possibly show relations between measures for clone classes and levels of coverage and precision. Such relations could then be tested on other systems, including those written in other languages.

Finally, it would be interesting to study how other techniques, especially PDG-based clone detectors [13, 12], are capable of identifying crosscutting concern code. Concerns like those included in our experiment consist of a large number of semantically equivalent code fragments. Due to the close match between PDG representations and the semantics of the program, we expect PDG-based clone detectors to perform well at detecting such crosscutting concern code. The experiment has shown that we can already identify large portions of crosscutting concern code by means of syntactical techniques (token-based and AST-based), mainly due to coding conventions and idioms used by developers. It will be interesting to see whether PDG-based techniques will perform better at finding crosscutting concern code in the same system, and also in systems which have not been implemented using the same coding conventions and idioms.

9. Conclusion

First, our results confirm the belief that crosscutting functionality is often 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 30% 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 the suitability of two clone detection techniques for the identification of code belonging

to crosscutting concerns. To that end, we manually identified four specific concerns in an industrial C application, and analysed the concern coverage and precision obtained by clone classes yielded by two clone detection tools. The results show that code belonging to concerns like parameter checking and memory error handling is identified very well by both clone detection tools, while error handling and tracing concerns are more problematic.

Acknowledgements 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).

References

[1] Brenda S. Baker. On finding duplication and near-duplication in large software systems. In *Second Working Conference on Reverse Engineering (WCRE'95)*, pages 86–95, Los Alamitos, California, 1995. IEEE Computer Society Press.

[2] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance (ICSM'98)*, pages 368–377. IEEE Computer Society Press, 1998.

[3] Silvia Breu and Jens Krinke. Aspect mining using dynamic analysis. In *GI-Softwaretechnik-Trends, Mitteilungen der Gesellschaft für Informatik*, volume 23, pages 21–22, Bad Honnef, Germany, May 2003.

[4] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In *Proceedings of the Joint European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9)*, pages 88–98, June 2001.

[5] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *Proceedings of the International Conference on Software Maintenance (ICSM'99)*, pages 109–118, September 1999.

[6] Ran Ettinger, Mathieu Verbaere, and Oege de Moor. Slicing Based Refactoring in Eclipse. Technical report, Oxford University Computing Laboratory, 2004.

[7] W. G. Griswold, Y. Kato, and J. J. Yuan. AspectBrowser: Tool Support for Managing Dispersed Aspects. Technical report, Department of Computer Science and Engineering, University of California, San Diego, 1999.

[8] Jan Hannemann and Gregor Kiczales. Overcoming the prevalent decomposition in legacy code. In *Proceedings of the ICSE Workshop on Advanced Separation of Concerns*, Toronto, Canada, May 2001.

[9] R. C. Holt. Binary relational algebra applied to software architecture. Technical Report 345, Computer Science Research Institute, University of Toronto, March 1996.

[10] J.H. Johnson. Identifying redundancy in source code using fingerprints. In *Proceedings of the IBM Centre for Advanced Studies Conference*, pages 171–183, 1993.

[11] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):645–670, July 2002.

[12] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the 8th International Symposium on Static Analysis*, pages 40–56. Springer-Verlag, 2001.

[13] Jens Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the Eight Working Conference On Reverse Engineering (WCRE'01)*, pages 301–109. IEEE Computer Society Press, October 2001.

[14] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of International Conference on Software Maintenance*, pages 244–254. IEEE Computer Society Press, November 1996.

[15] Filip Van Rysselberghe and Serge Demeyer. Reconstruction of successful software evolution using clone detection. In *Proceedings of the Sixth International Workshop on Principles of Software Evolution (IWPSE'03)*, pages 126–130. IEEE Computer Society Press, September 2003.

[16] David Sheperd, Emily Gibson, and Lori Pollock. Automated mining of desirable aspects. Technical Report 4, Department of Computer and Information Sciences, University of Delaware, Newark, DE 19716, 2004.

[17] Peri Tarr, Harold Ossher, William Harrison, and Jr. Stanley M. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st international conference on Software engineering*, pages 107–119. IEEE Computer Society Press, 1999.

[18] A. Walenstein. Problems creating task-relevant clone detection reference data. In *Proceedings of the Working Conference on Reverse Engineering (WCRE'03)*, pages 285–294. IEEE Computer Society Press, 2003.

[19] C. Zhang and H.-A. Jacobsen. A Prism for Research in Software Modularization through Aspect Mining. Technical report, Middleware Systems Research Group, University of Toronto, 2003.