Recovering System Specific Rules from Software Repositories

Chadd C. Williams
Department of Computer Science
University of Maryland
chadd@cs.umd.edu

Jeffrey K. Hollingsworth
Department of Computer Science
University of Maryland
hollings@cs.umd.edu

Abstract

One of the most successful applications of static analysis based bug finding tools is to search the source code for violations of system-specific rules. These rules may describe how functions interact in the code, how data is to be validated or how an API is to be used. To apply these tools, the developer must encode a rule that must be followed in the source code. The difficulty is that many of these system-specific rules are undocumented and "grow" over time as the source code changes. Most research in this area relies on expert programmers to document these little-known rules. In this paper we discuss a method to automatically recover a subset of these rules, function usage patterns, by mining the software repository. We present a preliminary study that applies our work to a large open source software project.

1 Introduction

Static analysis of source code has been used very successfully to locate bugs in software. One of the most successful applications of static analysis to find bugs has been tools that look for violations of system-specific rules in the source code. Source code must adhere to a large number of rules that describe how data should be handled, how to interact with objects or APIs and how to use functions safely. Violations of these system-specific rules are often a source of error [5].

The difficulty with these rules is that they are implicit and dynamic. As the source code changes new rules are added and old rules are removed. When functions are added to an API a new set of rules must be followed that describe how they are to be used. It is challenging for the developers of a widely distributed project to keep track of the rules the code must follow. This task is complicated by the fact that many of these rules are not documented as they are created, or are only documented in a CVS commit message or an email on a developer mailing list.

This leaves the project to rely on developers learning these rules in a number of unsatisfactory ways. For example, senior developers relating the rules that they know to new developers, developers searching CVS commit messages and mailing lists when they have a question or code reading. New developers are not the only ones to suffer. Senior developers need to keep up on the rules being added and removed from the source code.

In this paper we propose recovering these system-specific rules by studying the changes made to the source code. We specifically focus on rules that describe function usage patterns, how functions should be invoked in relation to each other. We believe that these usage patterns can shed light on how an external API should be used or how internal functions should interact. We have developed a tool that analyzes each version of a file in the software repository and determines what new function usage patterns are introduced in subsequent versions of each file.

2 Related Work

There has been ample research in the area of detecting violations of system-specific rules to identify bugs. One such system, metal [2], allows the user to supply patterns to match against the source code and flag as warnings. The patterns the developer supplies are encoded via state machines that are then applied to the source code. This system has been used to find a large number of errors (500) in real software projects. The metal system was also used to try to infer system specific patterns that should be checked [3]. While Engler, et al., look only at the current source code, our work focuses on looking at the changes made to the source code over time and what system specific rules these changes highlight.

Work has also been done to validate the notion that violations of system-specific rules cause a significant number of the errors seen in software [5]. Matsumura, et al., describe a case study that shows 32% of failures detected during the maintenance phase of a software project were due to violations of implicit code rules. The implicit rules used to check the source code were generated by 'expert' programmers.

The need for information sharing in large, distributed open source software projects has been studied. Gutwin, et al., studied the need for *group awareness*, knowledge about who is doing what is the project [4]. One of the aspects of awareness they describe is *feedthrough*, which is defined as observations of changes to project artifacts to indicate who has been doing what.

This work was supported in part by DOE Grants DE-FG02-93ER25176, DE-FG02-01ER25510, and DE-CFC02-01ER254489 and NSF award EIA-0080206.

There has also been work on identifying frequently applied changes to source code through mining the software change history [7]. Rysselberghe and Demeyer state that frequently applied changes can be used to study how software maintenance proceeds and to suggest solutions to future problems. They look for both system specific change patterns and more general patterns. While our work studies the state of the code after a change is made, their work looks exclusively at the changes applied to the code.

Pinzger and Gall identify patterns to recover software architecture [6]. They use code patterns specified by the user, and data describing the associatations of these patterns, to reconstruct higher-level patterns describing the software architecture.

3 Function Usage Patterns

The system-specific rules that we are studying in this work are *function usage patterns*. We want to determine how functions are invoked with respect to each other, specifically which functions are often called in close proximity within the source code. Instances of these patterns in a software project build up a set of *relationships* between functions. We define an *instance of a function usage pattern* as a set of two particular function call sites such that the pattern template is satisfied. We will explore the relationship aspect in Section 5.3. Experience suggests that there are sets of functions that are smaller parts of the implementation of a larger conceptual goal that need to be invoked together. These functions may operate on common data, provide error recovery functionality or perform some type of pair-wise functionality like lock/unlock. The two specific function usage patterns we are looking for are the *called after* and *conditionally called after* patterns. The *called after* relation is simple, function X is called after function Y in the source code of some function Z. The *conditionally called after* pattern describes the case where function X is called after function Y, but its invocation is guarded by a conditional statement. These two function usage patterns are the only two that we investigated for our preliminary study. Figure 1a provides an example of the *called after* pattern. Figure 1b provides an example of the *conditionally called after* pattern. Each of these code snippets highlight one instance of a function usage pattern identified by our tool in the Wine source code [10]. The code snippets have been edited for clarity.

There are a number of other patterns that might be useful. For example, in Figure 1b the function `GetProcessHeap` is called and its return value is used as an argument to both `HeapAlloc` and `HeapFree`. This type of pattern involving dataflow is something we plan to study in the future. A similar usage pattern is evident in Figure 1a between the functions `BeginPaint` and `DrawIcon`.

```
HDC hdc = BeginPaint( hwnd, &ps );
if( hdc )
    DrawIcon( hdc, x, y, hIcon );
EndPaint( hwnd, &ps );
```

Figure 1a: Called After Pattern

```
mdi_cs = HeapAlloc(GetProcessHeap());
if (!mdi_cs)
    HeapFree(GetProcessHeap(), 0, cs);
```

Figure 1b: Conditionally Called After Pattern

4 Our Tool

Our tool is very simple and casts a very wide net in terms of the instances of patterns it finds. This gives us the freedom to put off making decisions on how to filter the data until later in the process. This is important as retrieving the data from the software repository and generating our results is the most computationally expensive aspect of this work.

We use the framework developed for our previous work in mining software repositories to manage the data from the CVS repository and the results produced by our tool [9]. In summary, the data from the CVS repository and the raw results are stored in a database.

The tool we have produced is merely a prototype to support this preliminary study. It is based on the Edison Design Group C parser [1]. The tool parses the source file and scans for function call sites. Within each function in the source file, two function usage patterns are applied to each function call site. For every function call site in a function, every other function call site located later in that function is involved with it in a *called after* pattern (unless the later call site is guarded by an conditional). For each instance of a pattern, the tool records the names of each function, the line numbers of the call sites and the name of the enclosing function. The same process is used to determine *conditionally called after* patterns, with a bit more analysis to identify which functions are guarded by conditionals.

4.1 Mining the Source Code Repository

When mining the software repository we are looking for an instance of a function usage pattern in a revision of a file, where that instance of the pattern did not exist in the revision immediately prior. We are looking for new instances of patterns entering the code. Specifically with this tool we are looking for either a *called after* or *conditionally called after* pattern that did not exist in the previous revision of the file. Note that we are doing this on a per file, rather than on a per function, basis.

4.2 Identifying New Instances of Patterns

Once the data is mined from the source code repository and stored in the database, we must analyze it to determine when a new instance of a pattern has been added to the source code. Since our tool casts such a wide net in identifying patterns we need some way to filter the data. We have chosen, as a simple heuristic, to only look at instances of patterns that involve function invocations that are separated by no more than 10 lines of source code. This heuristic was chosen with the notion that many functions in an API need to be invoked in quick succession and that error handling, a possible target for the *conditionally called after* pattern, usually happens in close proximity to the error producing function.

In the future, we plan on refining this heuristic to be based on a deeper analysis of control flow. For example, the entry and exit basic blocks of a function may contain some function pairs that perform some type of paired functionality (lock/unlock). The basic blocks before a control flow split and after a control flow union may contain function calls related in some interesting way. Also looking at the type of conditional may be interesting. The conditional of a *while* loop versus that of an *if* statement may provide an important distinction between the applications of the *conditionally called after* pattern.

4.3 Transitive Patterns

Currently the patterns we are searching for are binary. The specific patterns we are searching for may be transitive in some cases, allowing larger relationship to be created. If a call to function *foo* is often followed by a call to *bar*, which is often followed by a call to *zoo*, then a call to *foo* may often be followed by a call to *zoo*. This transitivity may or may not exist. The context in which *bar* follows *foo* may be different from the context in which *zoo* follows *bar*. We may find we need to add more context information to our tool to differentiate usage patterns for a particular context. Section 5.3 contains a discussion of how to visualize the patterns mined from the source code.

5 Wine Case Study

We have used our tool to mine the software repository for the Wine project to determine what types of patterns can be recovered [10]. Each revision of each file has been analyzed by our tool. All instances of patterns that our tool finds are recorded in a database, tagged with the file and revision in which the pattern appeared.

Our tool identified over 50 million instances of these two patterns in the software repository. There were over 2,175 unique instances of patterns that were added to the source code 10 or more times. Sixty-five unique patterns were added to the source code 100 times or more. Many of these 65 patterns dealt with functions that manage the heap or provide tracing or debugging functionality.

5.1 Called After Pattern

As shown in Figure 1a, this pattern involves two functions, one called after the other. It is very simple and our goal with this pattern was to identify chains of functionality that need to be performed together. Our tool identified a number of patterns of this type, 1,253 unique instances of this pattern that were added to the source code 10 or more times. Some of the patterns identified were obvious, and while these did not provide novel insight, they did provide evidence that our analysis was working as expected. As mentioned, many of the instances found involved the heap management functions. In the Wine source code, almost every function that manipulates their internal heap must first retrieve the heap for the current process via `GetProcessHeap`. Consequently, many heap manipulation functions such as `HeapAlloc` and `RtlAllocateHeap` are called in close proximity to `GetProcessHeap`.

Our tool also identified a number of patterns that represent a notion of *paired functionality*. These patterns include pairings such as `BeginPaint` and `EndPaint`, `GlobalLock` and `GlobalUnlock` and `EnterCriticalSection/LeaveCriticalSection`. Again, these instances of the pattern are mainly interesting to validate the results.

A more interesting instance of the pattern involves the functions `DeleteCriticalSection` and `HeapFree`. In this case, once a critical section object has been deleted, the memory allocated for that object needs to be deallocated. This data structure appears to always be allocated off the internal heap (we also found, as another instance of the pattern, `HeapAlloc` followed by `InitializeCriticalSection`) and the memory on the heap needs to be freed to do this. Another instance of the pattern is `LoadCursorA` and `RegisterClassA.` The latter function takes as a parameter a data structure representing a class. One field of that data structure must be initialized with the return from the function `LoadCursorA.`

It is instructive to look at the categories of functionality that are being discovered in instances of these patterns. Table 1 shows how many new instances of the *called after* pattern fall into a selected group of categories. The number of new instances is broken down by how many times a particular instance of a pattern was flagged as new during the software repository mining.

Table 1 shows that debug statements are heavily used in the Wine source code. There are 97 instances of function usage patterns that involve a debug function and were added to the source code at least 25 times. This

Category	New Instances		
	> 99	99 - 25	24 - 10
Debug	17	80	278
Heap	14	16	16
String Manipulation	3	41	153
GUI	3	22	271
Memory	7	28	19
Paired Functionality	0	8	39
Error Handling	0	9	30

Table 1: Function Pairing Categories for Called After

means that there are 97 functions that are called in close proximity to a particular debug function.

The instances of the pattern listed in the Heap category are instances in which both functions involved are part of the heap interface. There are a total of 46 instances found in the code, indicating that functionality provided by the heap interface may require a number of function calls.

The category Paired Functionality contains instances of the pattern where the invoked functions provide functionality that needs to surround some bit of code. This includes such function pairings as `BeginPaint/EndPaint` and `GlobalLock/GlobalUnlock`. Eight such instances were added to the code between 25 and 99 times. Many of these instances involve some type of synchronization.

5.2 Conditionally Called After Pattern

The *conditionally called after* pattern is shown in Figure 1b. Our goal with this pattern was to see whether or not adding a small amount of control flow context to the pattern would help to elicit more interesting patterns. We expected this pattern to be able to identify error handling code and debugging idioms, instances of code where the second function is only called if the first function fails. Many of the instances of this pattern our tool identified supported this expectation. Our tool found 922 unique instances of this patterns that were added to the source code 10 or more times.

One of the instances involved the function `RegQueryValueExA` being conditionally called after `RegOpenkeyA`. In this case, the function `RegOpenKeyA` may or may not find a key in the registry. If it is successful the value can be queried. The insight here is that the developer cannot assume a key exists and should do the proper error checking to ensure that it was found properly.

Another interesting instance of this pattern is conditionally calling `SetLastError` after calling `HeapAlloc`. This instance of the pattern describes how errors should be propagated in the code. Table 2 shows how many new instances of the *conditionally called after* pattern fall into a selected group of categories based on functionality.

Category	New Instances		
	> 99	99 - 25	24 - 10
Debug	14	95	341
Heap	7	8	11
String Manipulation	0	25	121
GUI	0	3	94
Memory	0	19	17
Paired Functionality	0	6	26
Error Handling	0	3	34

Table 2: Function Pairing Categories for Conditionally Called After

5.3 Visualization

While the patterns we are searching for are binary, the functions involved may be part of many different instances of the pattern. Because of the type of patterns we are searching for, two functions that are each involved separately in an instance of a pattern with a common third function may themselves be related. This serves to build up a web of relationships, similar to those studied in the area of social networks. We have used a social network viewer, TouchGraph LinkBrowser [8], to explore the relationships between functions. Figure 2 shows the neighborhood of the network centered on `BeginPaint` and `EndPaint`.

Looking at this network graph gives quick insight into the functions that are invoked in close proximity to both `BeginPaint` and `EndPaint`. The function `BeginPaint` and `EndPaint` are used to wrap access to drawing functionality. We expect functions that provide this functionality to be found in instances of the *called after* pattern with either or both of these functions. The network in Figure 2 shows this clearly. We can see that `SetTextColor` and `GetClientRect`, for example, are attached to each of these functions. Further, the thin end of the edge is attached to the function which is called after the function at the thick end of the edge. We can see that `GetClientRect` is called after `BeginPaint`, and `EndPaint` is called after `GetClientRect`.

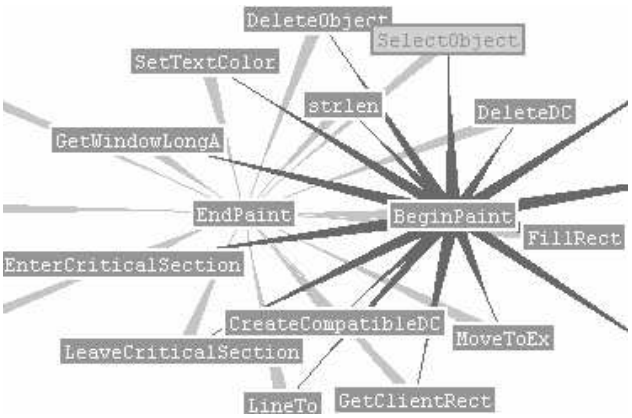

Figure 2: Social Network for BeginPaint/EndPaint

6 Why Mine the Full Repository?

We have chosen to mine each revision of each file to obtain a finer level of detail about changes made to the software. Since we gather data on what instances of patterns were added at each CVS transaction, we can investigate how instances of patterns entered the source code. Instances that are added to the source code steadily over time (over a large number of CVS transactions) may indicate a very important, frequently used pattern or a pattern that causes confusion among developers. On the other hand, patterns that are added to the source code in a relatively small number of CVS transactions may indicate refactoring. Determining the profile of how a pattern is added to the code may be useful in deciding the importance of that pattern, how to apply the instance in the future or how likely the pattern is to be misused by developers.

7 Future Work

The work we have presented here is still in its early stages. We have looked at only one software repository, and have only searched for instances of two patterns. In the future we will expand the number and complexity of patterns we search for and apply this technique to more software projects. We also do not track removed patterns. Knowing what patterns have been removed from the code could be useful in keeping an up-to-date list of important patterns in the project.

Mining the software repository of the Wine project has produced an enormous amount of data, a total of over 50 million instances of these two patterns were found in the repository. As we continue to work with this data we will need to find better ways of filtering out the more important, or more likely to be important, patterns. Currently our filter is based on the distance between, in terms of lines of code, the call sites of the two functions in the pattern. Clearly there is room for improvement. A filter that takes into account the files or directories the called functions (or the calling function) reside in may be useful in pulling out usage patterns of functions in the same module. Filters based on control flow graphs and deeper analysis of conditionals will provide more context as to the surrounding source code. Dataflow analysis as well will provide more context and may serve to provide a stronger link between two function calls. Finally, we need to not only think about patterns in terms of function calls. Patterns based on how data is accessed in a function, what parts of a structure need to be initialized or updated, need to be investigated as well.

We also need to explore how to use the instances of these patterns that are mined from the software repository. Providing these instances of patterns to a knowledge repository or as an appendix to a developer's guide may be a useful way to inform developer's of the system-specific rules the source code. Potentially more interesting is the use of instances of these patterns to automatically identify problems in the code. This may be done by feeding the rules into static analysis tools that identify violations of the rules in the source code.

8 Conclusions

In this paper we have demonstrated how system-specific rules, in this case function usage patterns, can be recovered from source code change histories. We have run a preliminary study to recover such rules from a large, open source software project. This study has recovered a number of interesting and non-obvious rules that we think are critical for developers to understand and follow.

9 References

[1] Edison Design Group, http://www.edg.com/cpp.html

[2] Engler, D., Chelf, B., Chou, A., Hallem, S., Checking System Rules Using System Specific, Programmer-Written Compiler Extensions. *In Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, San Diego, CA, October 2000.

[3] Engler, D., Chen, D. Y., Hallem, S., Chou, A., Chelf, B., Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code, In *Proceedings of the ACM symposium on Operating Systems Principles*, Banff, Canada, Oct 2001.

[4] Gutwin, C., Penner, R., Schneider, K., Group Awareness in Distributed Software Development, In *Proceedings of ACM Conference on Computer Supported Cooperative Work*, Chicago, IL, Nov 2004.

[5] Matsumura, T., Monden, A., Matsumoto, K., The Detection of Faulty Code Violating Implicit Coding Rules, *Proceedings of the International Workshop on Principles of Software Evolution* (IWPSE '02), Orlando, FL, USA, May 2002.

[6] Pinzger, M., Gall, H., Pattern-supported architecture recovery. In *Proceedings of the International Workshop on Program Comprehension* (IWPC'02), Paris, France, June 2002.

[7] Rysselberghe, F., Demeyer, S., Mining Version Control Systems for FACs (Frequently Applied Changes), *Proceedings of International Workshop on Mining Software Repositories* (MSR '04), Edinburgh, Scotland, UK, May 2004.

[8] TouchGraph LinkBrowser, Available online at http://touchgraph.sourceforge.net

[9] Williams, C. C., Hollingsworth, J. K., Bug Driven Bug Finders, In *Proceedings of International Workshop on Mining Software Repositories* (MSR '04), Edinburgh, Scotland, UK, May 2004.

[10] Wine, Available online at http://www.winehq.org