Automated Substring Hole Analysis

Yoram Adler, Eitan Farchi, Moshe Klausner, Dan Pelleg,
Orna Raz, Moran Shochat, Shmuel Ur, Aviad Zlotnick

IBM Haifa Research Lab, Haifa, Israel
{adler, farchi, klausner, dpelleg, ornar, morans, ur, aviad}@il.ibm.com

Abstract

Code coverage is a common measure for quantitatively assessing the quality of software testing. Code coverage indicates the fraction of code that is actually executed by tests in a test suite. While code coverage has been around since the 60's there has been little work on how to effectively analyze code coverage data measured in system tests. Raw data of this magnitude, containing millions of data records, is often impossible for a human user to comprehend and analyze. Even drill-down capabilities that enable looking at different granularities starting with directories and going through files to lines of source code are not enough.

Substring hole analysis is a novel method for viewing the coverage of huge data sets. We have implemented a tool that enables automatic substring hole analysis. We used this tool to analyze coverage data of several large and complex IBM software systems. The tool identified coverage holes that suggested interesting scenarios that were untested.

1. Introduction

Code coverage may be presented at several granularities. An example is starting with directories, then going through files, then functions, and finally source code statements. At each level we can have color-coded representations of the coverage. Presenting coverage data according to the hierarchical structure of the source code enables drilling down from directories to lines of code. However, the missing coverage is frequently not classified according to the predetermined hierarchy. For example, the code relating to execution on specific hardware platforms, to error paths, or to the handling of specific data types may cut across multiple elements of the hierarchy. We have seen cases where error paths that constitute more than twenty percent of the source code were not tested at all. However, this fact was not obvious from the coverage report of the system tests because the error handling code was evenly spread throughout the code. In addition, searching through the predetermined hierarchy is often tedious as the hierarchy might not divide the coverage tasks evenly.

Substring hole analysis aggregates and presents coverage information based on semantic similarities between source code elements, such as function names, rather than on the hierarchical structure of the source code. Semantic similarities between source code elements are determined by analyzing the names of the elements and identifying substrings that are common to multiple names. Test coverage information is aggregated and presented per substring rather than per source code element. Because the names given to software source code elements are usually indicative of their semantic meaning, source code elements with similar names are often associated with a common topic or context. As such, aggregating and presenting coverage information per substring provides the user with insight on aspects of the source code that lack coverage.

Reporting holes (i.e., coverage information per substrings) is challenging. There are many possible substrings, but they are often meaningless. Furthermore, users are typically willing to examine only a few results, so effective ranking of holes is crucial. However, different factors influence the risk that a hole represents. Examples include the number of coverage tasks in a hole and the percentage of coverage of a hole. These are similar to the support and confidence measures that are used in creating association rules [2]. These are multiple dimensions that the tool collapses into a linear order.

We ran substring hole analysis on several large IBM systems. The tool outperformed manual inspection by domain experts. While substring hole analysis is subjective by nature, our tool provides useful results and even suggests areas that lack coverage that the domain experts overlook. The tool is part of our coverage analysis tool FoCuS [1].

2. Definitions and tool description

A coverage task is identified by a string. A coverage task is covered if the code segment that it represents is executed.

The **input to the substring hole analysis tool** is a set of coverage tasks presented as character strings and the number of times that each task was covered. The coverage tasks are usually names of functions. For initial reports, it is sufficient to have a binary counter for each task indicating whether it was covered. An example of a single coverage task is "Exception.firm.io, 0".

A **hole** is defined by a given string. A hole is the set of coverage tasks for which it is a substring.

The **output of the substring hole analysis tool** is a ranked set of holes along with coverage data for each hole. For example, if a line of output in the coverage report for functions is "Exception, 912, 907", it means that there are 912 functions whose names contain the "Exception" substring, 907 of which were not covered.

Suggesting holes consists of two main challenges. Section 2.1 provides details of how our tool deals with the first challenge of identifying holes. Section 2.2 discusses the second challenge of prioritizing the identified holes for display. Section 2.3 briefly discusses tool parameters that influence the ranking schemes applied by the tool.

2.1. Identifying holes

Identifying holes means defining which substrings to examine. Naively, all possible substrings would be considered. However, we find it effective to prefer strings that have a semantic meaning. This heuristic eliminates much noise for the user and improves the tool's performance.

Fortunately, most programming styles make a distinction between words in names, either by introducing a delimiter such as '_', or by changing the letter case. The following rules define substrings to consider:

1. Characters are classified as digits, delimiters, uppercase letters, and lowercase letters.
2. Scanning an identifier from the left, a new substring starts whenever the characters change classes.
3. An exception to #2 is that the last of a sequence of uppercase letters moves to the beginning of the next substring if it starts with a lowercase letter.

This approach has an additional advantage of reducing computation time by avoiding overlapping substrings.

The tool also supports defining holes using two, possibly disjoint, substrings. We found this useful in detecting holes in operations that were uncovered by one component but not by another. E.g., "Cache*Callback" functions may be covered, but "Disk*Callback" functions may be uncovered. If "Disk", "Cache", and "Callback" by themselves are not holes that will be ranked highly, two substrings are needed. Our experiments did not show a need for combining more than two substrings.

2.2. Ranking holes

Typically, users only consider the first few holes, so the order of presentation is very important. This is not trivial: Is a hole of 900 out of 1000 more important than 100 out of 100? When "Exception" defines a hole of 500 out of 520 and "Exception.firm" defines a hole of 450 out of 450, which of them do you show to the user (further, assume that the string "Exception.soft" has coverage of 50 out of 70)?

There are a number of considerations to take into account when ranking holes:

- The number of tasks the hole represents
- The percentage of uncovered tasks
- The existence of similar holes
- Whether the hole cuts across the structured hierarchy of the program
- The length of the substring that represents the hole
- Whether it is possible to detect semantically meaningful substrings (see discussion in Section 1)

We have come up with a number of ranking schemes, implemented them all, and are currently studying and evaluating them. We believe that there is no single ranking scheme that is best. Our experience indicates that different users have different preferences.

2.3. Setting parameters

The tool supports various hole ranking schemes. A user may set minimum thresholds that influence the ranking scheme. Alternatively, a user may choose not to change the tool defaults. The user may set the upper and lower limits on the length of the substring that embodies the hole. Usually the minimum is between four and six, as lower than that may not be a meaningful name. The user may choose a minimum size of a hole, i.e., the number of functions that contain this substring. For example, if the user chooses 10, the tool does not consider any substring that belongs to less than ten functions. The last threshold is the percentage of the uncovered functions. We have different opinions and the number chosen is between 70% and 95%. The lower the percentage the more holes the tool displays. Holes that pass the thresholds are sorted according to one of five sorting functions. For example, the tool's default sort is to sort first on the ratio between the number of uncovered tasks per substring and the square root of Total—the number of functions containing the

substring (sqCov), then, in case of equality, further sort by Total; and finally, if still equal, sort by Length—the length of the substring. In a nutshell, this ranking is based on using a binomial distribution for the significance of not covering the hole's substring.

3. Summary

Substring hole analysis may be viewed as a post processing stage that requires only generic coverage data. Therefore, it is independent of the software's programming language, platform, and coverage data collection tool. This is in contrast to aspect oriented programming [0] that weaves cross cutting concerns based on the programming language.

We have used our substring hole analysis tool to analyze large quantities of coverage data from C, C++ and Java IBM software systems. The tool assisted users in identifying missing test scenarios.

Elsewhere [0] we discuss in detail the algorithms and heuristics the tool uses, and provide both experimental results and a mathematical framework for comparing hole ranking heuristics.

4. References

[1] FoCuS Code and function coverage tool **http://www.alphaworks.ibm.com/tech/focus**.

[2] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules", *Proc. of the 20th Int'l Conference on Very Large Databases*, Sep. 1994.

[3] Kiczales et al. "Aspect-Oriented Programming". *Proceedings of the European Conference on Object-Oriented Programming*, 1997, vol.1241. pp. 220–242.

[4] Yoram Adler, Eitan Farchi, Moshe Klausner, Dan Pelleg, Orna Raz, Moran Shochat, Shmuel Ur and Aviad Zlotnick. "Advanced Code Coverage Analysis Using Substring Holes". 2009. Submitted.

A. Tool Demo

We demo the substring hole analysis tool on a large IBM software system. The software is written in C++ and has 76,715 functions. The software is multiplatform. The particular test suite that was used for measuring coverage executes only on a subset of these platforms.

Figure 1 shows the entry screen of the substring hole analysis tool. In this screen the user sets parameters that may influence the tool's ranking of holes (as discussed in Section 3 above). We used the tool's default parameters as Figure 1 shows. The analysis is done on function names. The minimal length of substrings to consider is four characters. The maximal length of substrings to consider is thirty characters. The tool should only show holes that are at least 95% uncovered and contain at least ten coverage tasks. In the report we want to see no headers and only substrings.

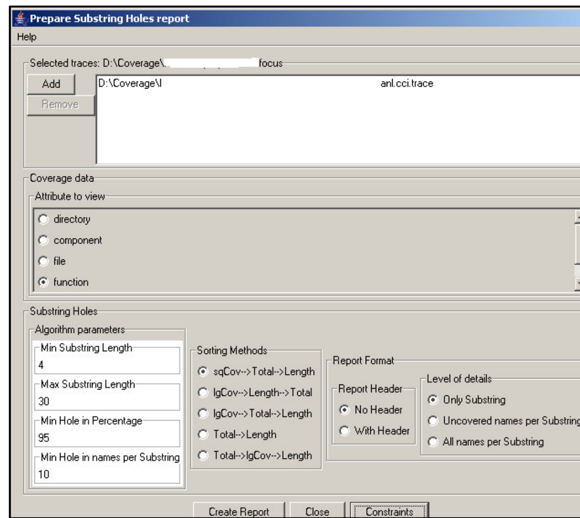

Figure 1 Substring hole analysis entry screen

Substring Holes Report - [D:\Coverage\FDPR\fdprSpec2006.focus\SubstringHolesSubs2008-

Font — Name: Courier New — Size: 16 — Bold — Italic

	Total	Uncovered	Sub-string
1	36709	36697	AbsAsm
2	36723	36697	Asm::
3	30187	30182	zSeries
4	18764	18726	Decomposer
5	14339	14124	Builder
6	12586	12551	::PPC
7	12443	12246	Instruction
8	8607	8604	Assembler
9	8566	8552	Set::
10	7808	7804	z*::
11	7239	7153	Name
12	6701	6692	Instructions
13	7006	6742	Type
14	5846	5835	Class
15	4980	4753	_d-const
16	4490	4315	Data
17	3461	3389	Instr
18	3082	3063	HLR*::
19	3082	3063	HLR::

Figure 2 Substring holes on all data (over 70,000 functions)

We applied substring hole analysis in two stages. First, we ran the tool on the entire data. Figure 2 shows the holes that the tool outputs. This output contains the expected holes, for substrings that are related to platforms that were not tested such as zSeries and Spe. In addition, the output contains holes that are related to

components that are not executed on the platforms that were tested. An example is all the holes related to Asm. Naturally, the coverage (function level) of the test suite on the entire code base is very low: 10.5%. The second stage in the analysis was to output holes that are based only on data the user cares about. To achieve that we applied constraints, a feature supported by the tool, on the coverage data.

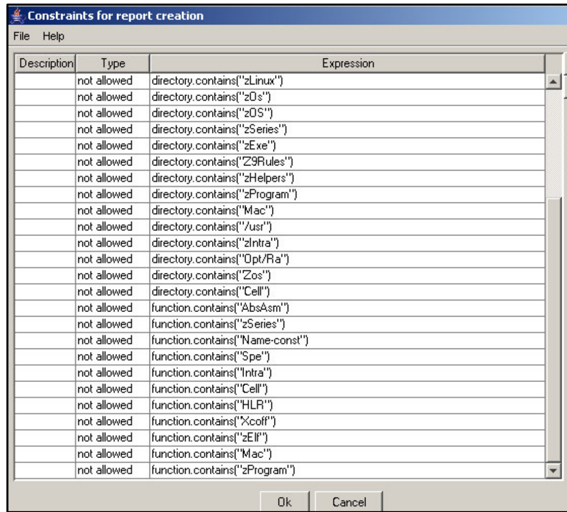

Figure 3 Constraints for eliminating data

Figure 3 shows some of the constraints we entered into the tool. An example of a constraint is not allowing coverage tasks that contain the string "zSeries" in the function name. After applying the constraints, the data contains 22,260 functions and the coverage over these functions is 35.7%.

Figure 4 shows the results of running substring hole analysis on the filtered data. An example of interesting holes are items related to Ppc. Figure 5 shows a drill down view on functions whose name contains 'Ppc' followed by any characters and then by '::~'. It turns out that the software has code for every instruction in the instruction sets it supports, such as the ppc instruction set. However, the test suite only contains tests for a small subset of these instructions, the ones that are most interesting. Presumably, unit tests were written to check all the other instructions. Finding the hole suggests that it may be a good idea to add these tests to the system test regression. Having these tests in the regression may reduce the risk of problems related to these instructions.

Another example of an interesting hole is Static, Helper, and Linkage shown in Figure 4. The test suite does not contain tests that cover the related component.

Notice that low coverage is typical during system test. This is one of the factors that make it difficult to get useful information from coverage data. Substring hole analysis assists in identifying the interesting areas to strengthen out of all the areas that are uncovered. In general, we do not expect 100% coverage in system test.

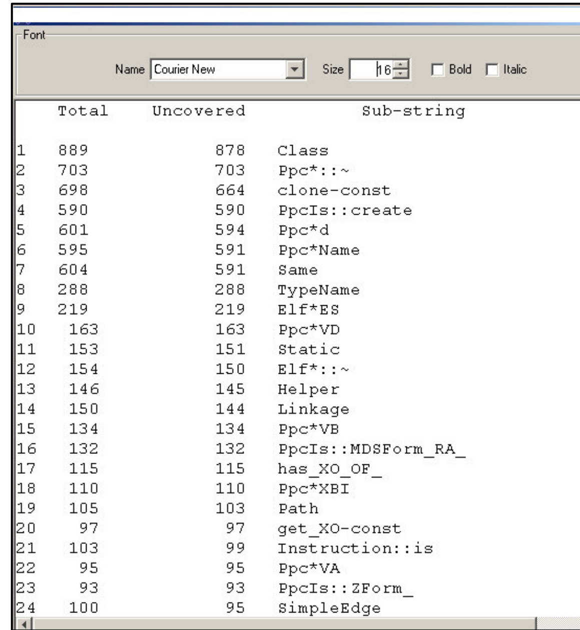

Figure 4 Substring holes on data with constraints (over 20,000 functions)

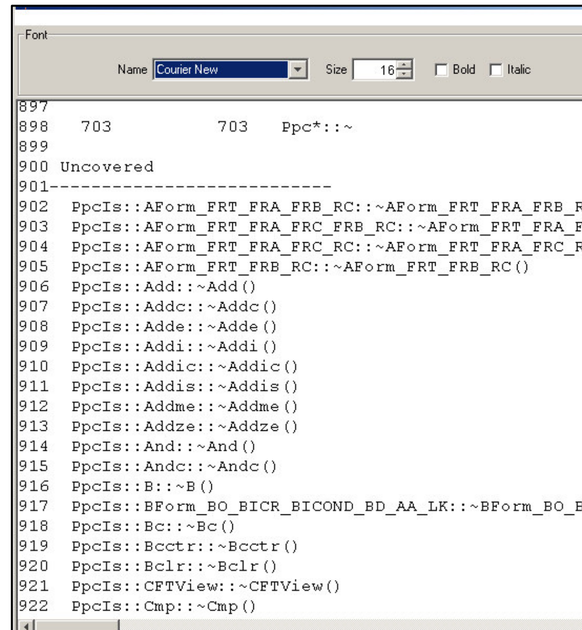

Figure 5 Drill down information for one hole Ppc*::~