

ConfViz: Supportive Configuration Visualization for Guaranteed Coverage Analysis

Angela Song-Ie Noh

Department of Computer Science, University of Maryland
College Park, USA 20742
angela@cs.umd.edu

Abstract. *Current software has many combinations of configurations to maintain the flexibility in extensions, modifications and supportive functions for software testing. However it is hard to explore all combinations of configuration settings followed by the increase of number of configuration options. Otter provides a symbolic evaluation for efficiently covering all these options by removing redundant or unnecessary combinations of settings. However it still maintains some limitations. It does not provide the overview of configuration options and its possible combinations since it is just showing the analysed results into the list view. Furthermore it is hard to glance the relationship between a list of relevant files and a configuration setting for a certain configuration combination setting. For certain lines of a code, it does not show the overview of strongly related combinations of configuration settings. All these results is possible to be found, but still need to be searched from the text output of Otter manually. To supplement those limitations, ConfViz provides the overview of such relationships by easy interaction. It displays the relevance of configuration options with files and lines of each file. It also shows the relationship between certain configuration settings and files. For each line, it provides the view representing the relationship between a set of combinations of configuration settings. This tool is tested on one configurable software system, 'vsftpd,' and shows well interaction of those features. Moreover it shows the tremendous decrease in scrutinizing a number of configuration combinations. Finally it will significantly help the configuration interaction testing as well as the understanding of software system behaviour.*

1. INTRODUCTION

The beauty of testing is found not in the effort but in the efficiency. Knowing what should be tested is beautiful, and knowing what is being tested is beautiful.

- Murali Nandigama,
Software engineer in Risk analysis, Mozilla

In software testing, knowing what to be tested in completeness manner is the most important objective. As a measure of testing completeness, code coverage is considered to be the most important factor of it. While

analysing the code coverage, it can find a dead code that is obsolete, which needs to be removed or to be added to another test suites to increase the code coverage. When modification of a program is done, this analysis helps finding appropriate test cases. If there is a part that is not exercised by a test suite this analysis helps find the designated part and make another test suite cover that part.

Today's software includes lots of configuration options. Thus if we use it wisely its analysis can help increase the code coverage. Current research on configuration analysis has following goals; one is to help understanding of program behaviour. Since there exist the lists of configuration variables with different settings, we can change its values and see how modification in a program – for example a change in a single source code; allows the partial testing for small part of a program. Next it helps the understanding of a correlation between some parts of the codes in certain files and its related configuration settings. This helps generating proper test cases as well as tracking bugs in a right spot.

Due to tremendous increase in configuration spaces current strategies focuses on sampling out the representative groups of configuration settings that covers the whole program instead of searching over the whole configuration lists. Combinatorial testing uses a t-way covering array so as to acquire good line coverage with the least amount of configuration settings [1]. However it still does not have commensurate paths and fault coverages [2, 3]. Besides it yet does not cover all possible combinations of configuration settings. It even omits some configuration settings that should be necessarily covered.

Hence to supplement this weakness, Otter evaluates every possible branch with all possible configuration settings by marking program values as symbolic. It updates the path condition with execution under certain assignment, and it explores all the paths until it is reachable. Then it creates possible configuration variable list. Otter projects all runs into four types of structural coverage. By this run, a partial guaranteed setting of configuration options is defined. Since the run covers all the lines, blocks, edge and condition, its partial settings are specified by those components [4].

Although this was developed for better interactions among each configuration option, it does not show the overview of relationship between relevant file lists and combination of configuration settings. It instead displays the analysed results of configurations for each file, which makes user hard to analyse and interact with different files with various different settings of configurations.

Thus this paper presents the visualization tool supports guaranteed coverage analysis by Otter. It not only shows the overview of configuration option relevance but also provides the relationship sketch between certain configuration settings and files. For better scrutiny in detail, it also presents the line numbers of each related code files associated with certain combination of configuration settings.

The remainder of this paper is composed as follows. Chapter 2 describes the process support for three kinds of evaluators. Chapter 3 sketches the overview of *ConfViz* tool using an example program and Chapter 4 describes the experiments. Related works are discussed in Chapter 5 and I conclude in Chapter 6.

2. PROCESS SUPPORT FOR EVALUATORS

ConfViz assumes that it is associated with Otter [5, 6]. Thus after the symbolic execution for guaranteed coverage, there are several outputs regarding with configuration analysis. Among those, our tool uses two of it. One is the list of configuration variables and the other is the line-by-line statement in a code analysis associating with configuration settings.

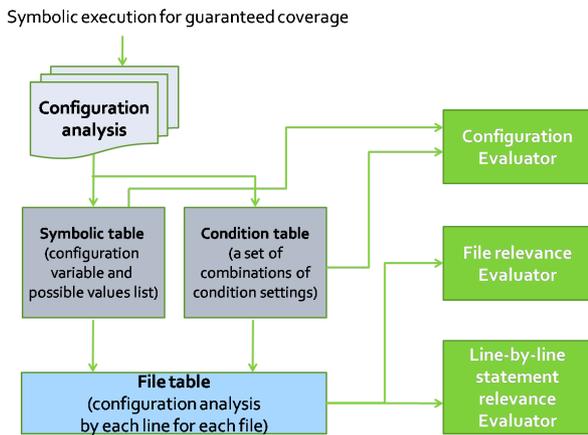


Fig. 1. System overview of *ConfViz* system

For analysing those two outputs this system creates three tables; a symbolic table, a condition table and a file table, as described in Fig.1. A *Symbolic_table* is a list of

configuration options and possible setting values for those. One configuration variable will have multiple possible values as an array and the first element of an array is set to a currently selected value. Its format will be stored in a following format. It firstly stores the configuration option and currently selected value. It then stores all possible values that can be assigned to that specified configuration variable. For the symbolic table for *cond_a*, it stores the configuration itself and its currently selected value along with all possible values, such as 0, 2048, 4029, and 65536.

Symbolic_table =
 $\langle \text{cond}_a, \text{selected_val}, 0, 2048, 4029, 65536 \rangle$

A *Condition_table* is composed of a set of combinations of condition settings. Each set will consist of multiple configuration variables and different values will be assigned. Here *Condition_1* consists of $(\text{cond}_a, 0) \wedge (\text{cond}_b, 1) \wedge (\text{cond}_c, 0)$. All conditional settings within same *Condition* are connected with AND predicate logic.

Condition_1 = $\langle (\text{cond}_a, 0), (\text{cond}_b, 1), (\text{cond}_c, 0) \rangle$
Condition_table =
 $\{ \langle (\text{cond}_a, 0), (\text{cond}_b, 1), (\text{cond}_c, 0) \rangle, \langle (\text{cond}_a, 0), (\text{cond}_d, 2048) \rangle \}$

For *Condition_table*, it consists of multiple combinations of *Conditions*. For the above example it has two sets of *Conditions* in it and those two conditions are connected with OR predicate logic. Thus *Condition_table* can be interpreted as follows.

Condition_table =
 $\{ (\text{cond}_a, 0) \wedge (\text{cond}_b, 1) \wedge (\text{cond}_c, 0) \}$
 $\vee \{ (\text{cond}_a, 0) \wedge (\text{cond}_d, 2048) \}$

A *File_table* is associating with these two tables, *Symbolic_table* and *Condition_table*, and a list of files. Thus it is constructed by a combination of those as the following form. One file is associating with multiple line-by-line statements and each line is connected to *Condition_table* that is a combination of multiple configuration settings.

File_table =
 $\{ \text{file_name}, \langle \text{line_num1}, \text{Condition_table}_A \rangle, \langle \text{line_num2}, \text{Condition_table}_B, \text{Condition}_C \rangle, \langle \text{line_num3}, \text{Condition_table}_A \rangle, \dots \}$

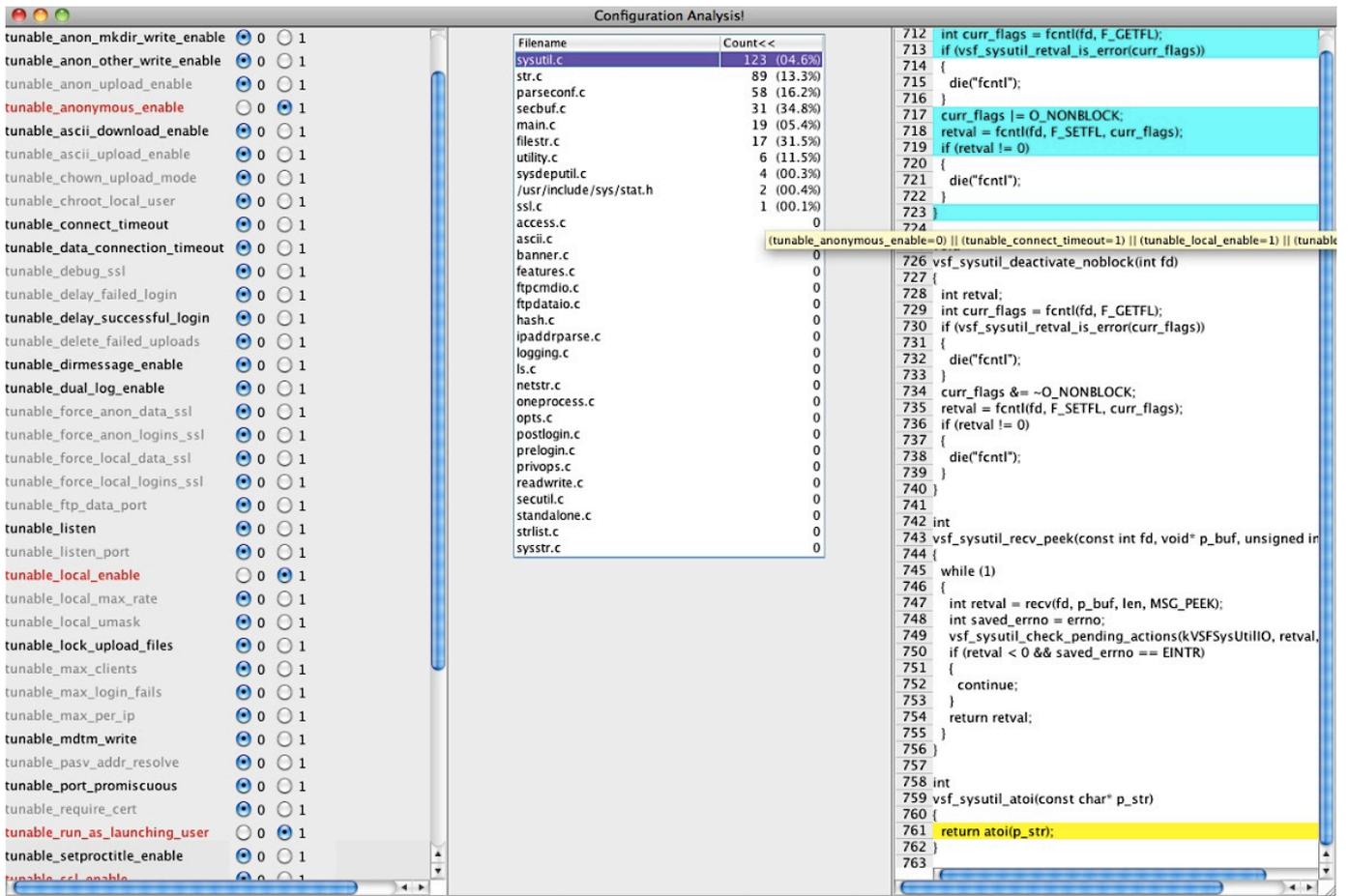


Fig. 2. Overview of ConfViz tool: this consists of three evaluators. The left one is a configuration evaluator and the middle one is a file relevance evaluator. And the right one is a line relevance evaluator.

To efficiently evaluate and manage those, we put three evaluators as described in Fig. 2 - a configuration evaluator, a file relevance evaluator and a line statement relevance evaluator – and their descriptions are briefly described.

2.1. Configuration Evaluator

The configuration evaluator initially loads a list of all configuration variable and their possible values. It then reads the actually used combination of configuration setting that is hit by lines in multiple files. This information is passed from Otter symbolic evaluator. By checking the uniqueness of combinations of configuration settings, we can observe how many configuration settings are actually used. Instead of examining over all possible configuration spaces, we can notice that only a small number of combinations of configuration settings are used. Through this evaluation we can check efficiency and completeness of code coverage.

2.2. File Relevance Evaluator

The file relevance evaluator checks whether specified configuration settings may hit by any line in any files. To check the relevance of a specified configuration setting, it counts the number of lines that may hit by specified configuration setting. Since counted number is in an absolute value so it cannot be a comparable measurement to the size of the file. Thus ConfViz system also provides the view of checking the percentage of hit lines to the whole number of lines in a file. Two different representations; that is in absolute number of related lines in each file and relative percentage associating with certain setting – provides the view to users easily checking which file is strongly related to such setting. When the users are interested in checking the file relevance by file names, then they can click on *Filename*, which lets sorting those alphabetically. If they are interested in ranking for the file relevance in specified configuration setting, then they can click on *Count*, then it will be ranked by ascending or descending order.

2.3. Line Relevance Evaluator

Some lines in a file may be hit by certain configuration combination settings. Thus one file will have multiple lines to be hit and each line can have one or more lines to be hit. For interactive understanding of this relationship, it directly refers to *File_table* in the *ConfViz* system. When clicking on one *Filename* with default configuration settings, it will show the relationship between a file and different combinations of configuration settings for designated file. If modifying the configuration settings from configuration evaluator and choosing the file from the file relevance evaluator, then it will only show the related lines in that file with defined configuration setting.

3. OVERVIEW OF CONFVIZ TOOL

3.1. Implementation

ConfViz is implemented in Java. The objective of this tool is easy interaction and visualization of relevant configuration setting and files. *ConfViz* consists of three main components, configuration variables, files and the lines. To represent those three components at a glance it has three perspective views. As described in *Fig. 2*, the left panel shows the configuration option list. Here the user can select values for certain configuration variables. All the choices are available by radio button, so user can choose only one setting for each configuration option. Middle panel shows the list of all files in a configurable program. If the file contains any line that may be hit by selected configuration settings chosen from the left panel, then the file relevance evaluator passes the relevance factor of each file. Right panel displays the actual source code file selected from the middle panel. It actually shows which lines of selected file are hit by specified configuration setting.

3.2. Interaction

ConfViz has five special features for better understanding of configuration analysis and its detail will be described as follows.

3.2.1. Actual usage of configuration variables

In practice there are a number of configuration variables for each program. Assume there are n configuration variables. Each variable can have equally *two* possible values to be selected. The size of configuration space that

should be examined in this case will be 2^n . However not all configuration combinations are actually used for checking the code coverage. Likewise developers do not need to test on all combinations of configuration settings for software testing. *Configuration evaluator* described in *Ch. 2.1* evaluates the actual usage of configuration variables to find out related configuration options and its settings. In this way obsolete configuration settings can be removed from the testing path branches. In *Fig. 3*, if multiple combinations of actually used configuration settings include the certain configuration option for at least once, then it will be presented in black. Otherwise it will be presented in grey, which means that developers do not need to check those configuration variables for combinations, since it has not been actually used. If one configuration option is selected, then the other related configurations that can form a combinations for *Conditions*, will turn into red. In this way user can easily refer the possible combinations of configuration settings.



tunable_anon_mkdir_write_enable	<input checked="" type="radio"/>	0	<input type="radio"/>	1
tunable_anon_other_write_enable	<input checked="" type="radio"/>	0	<input type="radio"/>	1
tunable_anon_upload_enable	<input checked="" type="radio"/>	0	<input type="radio"/>	1
tunable_anonymous_enable	<input type="radio"/>	0	<input checked="" type="radio"/>	1
tunable_ascii_download_enable	<input checked="" type="radio"/>	0	<input type="radio"/>	1
tunable_ascii_upload_enable	<input checked="" type="radio"/>	0	<input type="radio"/>	1
tunable_chown_upload_mode	<input checked="" type="radio"/>	0	<input type="radio"/>	1
tunable_chroot_local_user	<input checked="" type="radio"/>	0	<input type="radio"/>	1
tunable_connect_timeout	<input checked="" type="radio"/>	0	<input type="radio"/>	1
tunable_data_connection_timeout	<input checked="" type="radio"/>	0	<input type="radio"/>	1
tunable_debug_ssl	<input checked="" type="radio"/>	0	<input type="radio"/>	1
tunable_delay_failed_login	<input checked="" type="radio"/>	0	<input type="radio"/>	1
tunable_delay_successful_login	<input checked="" type="radio"/>	0	<input type="radio"/>	1
tunable_delete_failed_uploads	<input checked="" type="radio"/>	0	<input type="radio"/>	1

Fig. 3. Actual usages of configuration variables are differed by color; black, grey and red.

3.2.2. Relationship between configuration settings and files

As described in *Fig. 4*, there exist two relationships in configuration analysis. *Fig.4(a)* show the related combinations for each configuration variable. Since configuration settings do not work alone, but works with combination settings with other variables, one configuration variable will have multiple related combinations with other variables. This information is loaded from *Condition_table* and shows all *Conditions* in it. Thus when right-clicking on one configuration variable, then all possible combinations

tunable_dual_log_enable=1	tunable_anon_mkdir_write_enable=1	tunable_anonymous_enable=1	tunable_dual_log_enable=1	tunable_local_enable=0	tunable_ssl_enable=0	tunable_write_enable=1
tunable_force_anon	tunable_anon_other_write_enable=1	tunable_anonymous_enable=1	tunable_dual_log_enable=1	tunable_local_enable=0	tunable_ssl_enable=0	tunable_write_enable=1
tunable_force_anon	tunable_anonymous_enable=1	tunable_dual_log_enable=0	tunable_local_enable=0	tunable_ssl_enable=0		
tunable_force_local	tunable_anonymous_enable=1	tunable_dual_log_enable=1	tunable_local_enable=0	tunable_run_as_launching_user=0	tunable_ssl_enable=0	
tunable_force_local	tunable_anonymous_enable=1	tunable_dual_log_enable=1	tunable_local_enable=0	tunable_ssl_enable=0		
strlist.c	39 (22.4%)	232	void			
sysdeputil.c	34 (02.9%)	233	str_append_text(struct mystr* p_str, const char* p_src)			
secbuf.c	32 (36.0%)	234	{			
logging.c	27 (07.3%)	235	unsigned int len = vsf_sysutil_strlen(p_src);			
secutil.c	24 (19.2%)	236	private_str_append_memchunk(p_str, p_src, len);			
(tunable_anonymous_enable=1	tunable_local_enable=0	tunable_ssl_enable=0) (tunable_listen=1	tunable_setproctitle_enable=1	tunable_ssl_enable=0)		
netstr.c	22 (19.0%)	238				

Fig. 4. (a) Upper: Related combinations of configuration settings for selected configuration variable.

(b) Down: Groups of configuration combination that may be hit by specified line number.

Filename	Count<<
sysutil.c	127 (04.8%)
str.c	89 (13.3%)
parseconf.c	58 (16.2%)
main.c	37 (10.6%)
secbuf.c	31 (34.8%)
filestr.c	17 (31.5%)
sysdeputil.c	9 (00.8%)
utility.c	6 (11.5%)
/usr/include/sys/stat.h	2 (00.4%)
sysstr.c	0
strlist.c	0
standalone.c	0
ssl.c	0
secutil.c	0
readwrite.c	0
privops.c	0
prelogin.c	0
postlogin.c	0
opts.c	0
oneprocess.c	0
netstr.c	0
ls.c	0
logging.c	0
ipaddrparse.c	0
hash.c	0
ftpdataio.c	0
ftpcmdio.c	0
features.c	0
banner.c	0
ascii.c	0
access.c	0

Fig. 5. Sort the relevant file list by filenames and the number of relevant lines in each file

related to selected configuration option will be displayed. Each combination will be represented by different row. Users can choose among possible combinations based on their interest.

When selecting a file, multiple lines can be hit by one specified setting. At the same time other possible combinations of settings can also hit one line. To help understanding of this relationship, when users can move the mouse over on certain line number of a file. Then all of its possible combinations of configuration settings will be displayed. This function is well described in Fig.4.(b).

```

102 private_str_alloc_memchunk(p_str, p_src, len);
103
104
105 void
106 str_alloc_ulong(struct mystr* p_str, unsigned long the_ulong)
107 {
108     str_alloc_text(p_str, vsf_sysutil_ulong_to_str(the_ulong));
109 }
110
111 void
112 str_alloc_filesize_t(struct mystr* p_str, filesize_t the_filesize)
113 {
114     str_alloc_text(p_str, vsf_sysutil_filesize_t_to_str(the_filesize));
115 }
116
117 void
118 str_free(struct mystr* p_str)
119 {
120     if (p_str->p_buf != 0)
121     {
122         vsf_sysutil_free(p_str->p_buf);
123     }
124     p_str->p_buf = 0;
125     p_str->len = 0;
126     p_str->alloc_bytes = 0;
127 }
128

```

Fig. 6. Two types of configuration settings presented in different color; blue and yellow

3.2.3. Relevance of configuration setting to a list of files

To help understanding of developers in relationship between specified configuration settings and other components, the user should be able to briefly overview its relevance as a rank. Users may be interested in checking the relevance of certain file with certain configuration setting. In addition users may pay attention to the list of strongly related files for certain configuration setting.

For satisfying the user interest, sorting of the relevant file lists are available both by file name and by the number of lines hit by specified configuration settings for each file as Fig. 5 shows. For changing the order you can click either on *Filename* or *Count* so that each of them may be sorted in ascending or descending order. This sorting function helps users understanding the relevance between certain configuration settings and the files.

```

549 if (s_exit_func)
...
556 _exit(exit_code);
...
712 int curr_flags = fcntl(fd, F_GETFL);
...
713 if (vsf_sysutil_retval_is_error(curr_flags))
...
717 curr_flags |= O_NONBLOCK;
...
718 retval = fcntl(fd, F_SETFL, curr_flags);
719 if (retval != 0)
...
723 }
...
761 return atoi(p_str);
...
848 unsigned int result = 0;
...
849 int seen_non_zero_digit = 0;
...
850 while (*p_str != '\0')
...
852 int digit = *p_str;
...
853 if (!isdigit(digit) || digit > '7')
...
857 if (digit != '0')
...
859 seen_non_zero_digit = 1;
...
861 if (seen_non_zero_digit)
...
863

```

Fig. 7. Expand and Collapse the file line blocks

3.2.4. Two types of configuration settings

As described in Ch.2, *File_table* is composed of several lines of a file and each line is associated with multiple combinations of configuration settings. There exist two types of configuration settings. The first type is executable under a combination of several configuration settings. If there exists any combination of configuration settings, the line of a code will be displayed in blue. The second type happens when there exists a line hitting regardless of any option setting. In such case the combination is always represented as *true*. The line with such configuration setting will be shown in yellow. Fig. 6 shows the presentation of those two types of configuration settings in one file.

3.2.5. Expanding and Collapsing the relevant file line blocks

When relevant lines are scattered in one file, it is hard to see the overview of related code lines. For such case *ConfViz* provides the expanding and collapsing of the relevant file line blocks. If double clicking on the file name of interest, only the first line of the relevant blocks will be shown with the color. Fig. 7 shows the collapsed mode of the file line blocks. If double clicking on the file name, user can see the whole part of the file.

4. APPLYING THE TOOL TO THE DATA SET

The subject program for this study is *vsftpd*, which is one of the widely used secure FTP daemons and all files are written in C. It has multiple configuration options that can be set in system configuration files or command line parameters. Table 1 shows the subject program statistics.

TABLE I
SUBJECT PROGRAM STATISTICS

	<i>vsftpd</i>
<i>Version</i>	2.0.7
<i># Files</i>	77
<i># Lines (sloccount)</i>	4,112
<i># Analysed Configuration Variables</i>	49
<i># Analysed Configuration Options</i>	
<i>Binary</i>	48
<i>Integer</i>	1
<i># Possible Combinations of Conditions</i>	$2^{48} * 4 = 2^{50}$
<i># Analysed Combinations of Conditions</i>	2,691
<i># Distinct Combinations of Conditions</i>	44

The total number of files, lines and analysed results are extracted from [4]. From this table, we know that *vsftpd* has 77 distinct files and 2^{50} possible combinations of conditions to be examined. However after running *Otter*, it showed that there only exist 2,691 different conditions. Since there are redundancies in configuration combination sets, our configuration evaluator only counts the distinct combinations of conditions and it is only 44.

Two CS graduate students were selected as tester for this tool. I explain the goal of this tool for 15 minutes and received feedback on the usability and interaction design. I found out that this tool is superior for seeing the overview of the relevant files and individual lines in it for chosen configuration settings. In addition showing actual combinations of configuration settings as a tooltip was useful for choosing configuration options. However the total number of unique configuration options and possible combinations of configuration settings are not available, so this information should be displayed in a certain way. Expanding and collapsing the file line blocks is useful for seeing the overview, but after double clicking on the file name, whole parts of the source code is displayed instead of only showing that selected file line blocks. Thus this part should be modified. Sorting by the count is good for checking the relevant files, but there should also exist the

sorting by relevant percentages. Sometimes this feature can help user more in debugging.

5. RELATED WORK

Symbolic Evaluation. Configuration options affecting software runtime behavior is hard to be distinguished easily. One simple way to test configuration's effect is searching through code for each configuration options but it is not efficient. A symbolic evaluator, Otter [5, 6], takes unknown values, which may take on any values and tracks these values through the program. When these values determine the path condition, the program forks the execution and marks them as symbolic values. By repeating this process Otter can create all possible paths through the whole program based on these symbolic values. This path condition result can be represented in a tree structure, which is called as an execution tree.

These symbolic values are useful for executing a set of test cases. Single predicate with other arbitrary choices of values can even cover one test case in a branch. Thus showing the relationship between configuration options and its value in an interactive view can strongly help analyzing the runtime behavior.

The output from Otter is only represented as a list. It only has the list of each file name, line numbers and its corresponding configuration option settings, which forbids the easy interaction of changes between configuration options and related files in a program.

This study emphasizes the interaction of configuration settings, specified values and related files. By representing those different components in different panels, user can easily interact with desired settings and study its effect when the setting has been changed.

Software Visualization Tools. Software visualization is a method for graphically representing software's behavior, structure, execution and its evolution [7]. Before visualizing the task, it initially should decide the purpose of visualization task; in detail we should clarify whether this software is for showing the results of program analysis, software architecture, dynamic data acquisition, and inspection on program states or debugging. After clarifying the purpose of visualization, it can then work out how to present those.

Source Navigator [8] shows the related list of files with selection followed by wildcards. It also highlights different variables by different colors. It displays referenced functions by tree structure so that users can easily

understand the overall structure. Even though the tree structure shows the overview of its components in hierarchy it is not applicable for showing the overview when changing the relationship of configurations and source code itself. For example if there exist different combinations of configuration settings and lines in a code, it is hard to track it down.

Goose [9] extracts the entities and relationship of meta-model from source code in C/C++ and Java. Since it shows the close interaction and relationship among configurations in a graph-based view, the lines representing their relationship can be tangled when their relationship becomes more complex. Their connected edges can be overlapped in such case, so it may be hard to distinguish between them.

In general graph structure distracts the user interpretation, thus this study does not take this representation. Instead this paper takes the idea of showing the list with tree structure. When showing the relevant configuration combinations, the configuration's value interacts with the selected file intimately. When the configuration is set, related files list is showed up. When mouse over each line of the source code, its related configuration combination is shown up as a tool tip. Since each component, such as a configuration list, a file list and an actual selected source code, are represented in a different panel in the window, their relevance is represented clearly.

6. CONCLUSION AND FUTURE WORKS

This study arises from configuration analysis for improving the code coverage in software testing. As the size of configuration spaces grows it is hard to cover all spaces because of its combinatorial problem. To minimize the combination of configurations with improved code coverage, current study uses a *t-way covering array* [10, 11]. However it still does not cover all cases and even misses some cases that should be covered.

To overcome this current issue, *symbolic execution* checks all possible reachable program paths with arbitrary assignments. *Otter* is one of the systems analyzing the system runtime behavior as configuration setting changes. It yet does not tell the overview picture of relationship among configuration settings, files and lines of the code.

As a supportive tool, *ConfViz* visualizes these tasks. It provides the overview of relevance and relationship between certain configuration settings, files, and lines. Moreover this tool removes the obsolete combinations of configuration settings and omits the redundant combination sets. I expect

this will help developers' effort to scrutinizing all combinations of configuration settings.

All relevancies are represented in three different panels, but detail information is not shown, such as the total number of unique combinations of configuration settings and distinct configuration options are not shown. Tooltip is currently used to show the possible combinations for one setting, or to show the setting for one line in the file, but when it becomes long, it shows the inferior readability. Thus given the ability to continue developing this tool, the status panel that shows the detail information can be added at the bottom. In that way user need not to worry about the long lines of the tooltip.

Current sorting of the files are only based on filenames in alphabetical order and counts in ascending or descending order. However to diagnose the relevance correctly the sorting of percentage should be added.

Expanding and collapsing the related code blocks are good feature to see the overview in one file, but this feature should be applicable not only to the whole file, but also to the code line blocks. In that way user can easily interact between code lines blocks and the whole code with checking the relevancy.

For user interaction perspective, the user usually starts everything from the left panel to the right one. It initially selects the configuration settings, selects the relevant file and then sees the detail view inside the file. However user's workflow can start from the right panel to the left one or from the middle panel to the left one. Our tool is based on the common workflow, but still not fully supportive for the other workflow. If I can continue the research on this, the feature that allows user to select one file and checks all possible sets of combinations for configuration options and all related lines should be added. In such case, it is hard to remain the left panel as the current view, effective presentation of all possible combinations should be added further.

Moreover this system is now only tested on *vsftpd*. Later on, if it is possible, this can be tested on other configurable system and proves its efficiency with some usability testing.

ACKNOWLEDGMENT

I wish to thank Charles Song for helping me to get output files of Otter, a symbolic evaluator.

REFERENCES

- [1] M. B. Cohen, J. Snyder and G. Rothermel. Testing across configurations: implications for combinatorial testing, Workshop on advances in model-based software testing(A-MOST), Raleigh, North Carolina, pp. 1-9, 2006
- [2] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C.Patton. The AETG system: an approach to testingbased on combinatorial design. TSE, 23(7):437, 44, 1997
- [3] V. Ganesh and D. L. Dill. A decision procedure forbit-vectors and arrays. In CAV, July 2007
- [4] ElnatanReisner, Charles Song, Kin-Keung Ma, Jeffrey S. Foster, Adam Porter. 'Using Symbolic Evaluation to Understand Behavior in Configurable Software Systems', ICSE 2010
- [5] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death, CCS, pp. 322-335, 2006
- [6] P. Godefroid, N. Klarlund, and K.Sn. DART: directed automated random testing, PLDI, pp 213-223, 2005
- [7] Stephan Diehl, Software visualization; visualizing the structure, behavior and evolution of software, Springer, 2007
- [8] Source Navigator Documentation, <http://sourcnav.sourceforge.net/online-docs/index.html>
- [9] Oliver Ciupke. Automatic detection of design problems in object-oriented reengineering. In technology of object-oriented languages and systems - TOOLS 30, pages 18-32, Santa Barbara, CA, August 1999
- [10] C. Yilmaz, M. B. Cohen and A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces, IEEE Transaction on Software Engineering, 31(1), pp. 20-34, 2006
- [11] S. Fouche, M. B. Cohen and A. Porter. Incremental covering array failure characterization in large configuration spaces, international symposium on software testing and analysis (ISSTA), pp. 177-187, 2009