2011 22nd IEEE International Symposium on Software Reliability Engineering

# Impact Analysis of Configuration Changes for Test Case Selection

Xiao Qu, Mithun Acharya, Brian Robinson
Industrial Software Systems
ABB Corporate Research
Raleigh NC USA 27606
{xiao.qu, mithun.acharya, brian.p.robinson}@us.abb.com

*Abstract*— **Testing configurable systems, which are becoming prevalent, is expensive due to the large number of configurations and test cases. Existing approaches reduce this expense by selecting or prioritizing configurations. However, these approaches redundantly run the full test suite for the selected configurations. To address this redundancy, we propose a test case selection approach by analyzing the impact of configuration changes with static program slicing. Given an existing test suite *T* used for testing a system *S* under a configuration *C*, our approach decides for each *t* in *T* if *t* has to be used for testing *S* under a different configuration *C'*. We have evaluated our approach on a large industrial system within ABB with promising results.**

***Keywords- configuration testing; test case selection; program slicing; static impact analysis***

## I. INTRODUCTION

User configurable software systems — systems that can be customized through a set of options by the user — are becoming increasingly prevalent. When these systems are used with different configurations, for the same operations, the systems may behave differently as different parts of the system source code may be executed. As a result, new faults may be revealed [6][17][26]. For example, a conflict between IE7 and the Google toolbar was reported by many users in 2006: users of the IE7 browser suffered from the problem of losing the "*Open in New Tab*" option in the right-click menu [29]. To solve this problem, the users had to disable or uninstall the Google toolbar. With different configurations (i.e., with Google toolbar enabled or disabled), the system, IE7, behaves differently with the same operation (right clicking on a link). From a testing perspective, if IE7 was tested with the Google toolbar disabled, this incompatibility (hence, a fault) would not be detected. But, if IE7 was tested with the Google toolbar enabled, the fault would be detected.

Therefore, a configurable system should be retested when the system is running with a new, previously untested configuration. Configuration testing can be expensive because configurable systems usually have a large number of possible configurations and test cases. To reduce this expense, one can select (or prioritize) configurations from all possible configurations and select test cases for the selected configurations from an existing test suite. Configuration selection (prioritization) and test case selection are in fact two different dimensions of configuration-aware testing. Approaches exist for selecting [5][19] or prioritizing [17] configurations. However, these approaches run the full test suite for each of the selected configurations. This causes redundant testing, as different test cases can have exactly the same behavior or coverage with different configurations with no increase in the fault detection capability.

To address this problem, we propose an approach for selecting test cases for the pre-selected configurations of a system. Our approach assumes that a test suite already exists for the system under test and that there are no changes made to the system source code when testing with different configurations. To formalize, let an existing test suite *T* be used for testing a system *S* under a configuration *C*. Our goal of test case selection is to decide for each $t \in T$, if *t* has to be used for testing *S* under a different configuration *C'*. Our study results show that on average, only 20% of the test cases were enough to test a large system developed at ABB, when a configuration under test changes. Considering that large systems usually test hundreds or even thousands of configurations, our approach will lead to substantial savings.

A test case selection approach is *safe* if it always selects a test case $t \in T$ whenever the parts of the system source code executed by *t* are different under C and *C'* (i.e., the dynamic code coverage of *t* in *S* is different under *C* and *C'*) [22]. However, in practice, the coverage data of *t* under *C'* is unavailable unless we rerun the full test suite *T* under *C'*. Therefore, our approach statically approximates, while being conservative, the parts of source code where *t* executes differently in *S* under *C* and *C'*. Specifically, by analyzing the impact of configuration changes (i.e., the differences between *C* and *C'* mapped to the source code) through static *program slicing* [24], our approach selects test cases in *T* which covered parts of the source code that are statically impacted by the configuration changes. Our study results demonstrate that our static approximation of the dynamic coverage can lead to a safe test case selection. Moreover, unlike most coverage-based test case selection approaches [21][22], which only consider control flow, our approach considers both control and data flow for test case selection. Hence, our approach may select additional useful test cases.

The rest of the paper is organized as follows. In Section II, we describe our approach of test case selection with an illustrative example. In Section III, we describe the implementation of our approach. In Section IV, we present a study to evaluate our approach. In Section V, we present the results of our study. In Section VI, we discuss the limitations and threats to validity of our approach. In Section VII, we discuss the related literature. Finally, in Section VIII, we conclude and identify areas for further research.

## II. APPROACH

In this section, we describe our test case selection approach. In Section A, we describe the concept of configurable systems with an illustrative example. In Section B, we define the notations used to describe our approach. In Section C, we define the problem addressed by our approach. In Section D, we provide an overview of our approach. In Section E, we describe our approach in detail using the example we present in Section A. In Sections F-H, we discuss several considerations related to our approach such as the safety criterion of our approach and the opportunities for time savings with the static analysis used in our approach.

### A. Configurable Systems

A configurable system has various configurable *options* that control the system's execution. The specific execution of the system depends on the actual *values* supplied for these options. For example, *vim* is an open source editor which is configurable. In *vim*, if the value `ON` is assigned to one of the *vim*'s configurable options, `more`, *vim* pauses listing a file when the whole screen is filled. If `OFF` is assigned to `more`, *vim* continues listing a file until finished.

Given a configurable system, let us denote the set of $m$ configurable options by $P=\{P_1, P_2, P_3, ..., P_m\}$. A particular assignment of values to each configurable option forms a *configuration instance* (henceforth, we use the term configuration and configuration instance interchangeably). For each $P_i$, let $P_i^C$ represent the value assigned to $P_i$ in configuration $C$. Hence, the configuration instance $C$ can be represented by the set $\{P_1^C, P_2^C, ..., P_m^C\}$. An example configurable system, denoted by $S$, is shown in Figure 2, along with two test cases $T=\{t_0, t_1\}$. $S$ executes three functions ($f_1$, $f_2$, and $f_3$) with two configurable options $P=\{P_1, P_2\}$. Figure 1 shows two configurations $C=\{OFF, OFF\}$ and $C'=\{ON, OFF\}$ that are pre-selected for testing $S$. When $S$ is tested under configuration $C$, $t_0$ executes $f_2$. When $S$ is tested under configuration $C'$, $t_0$ executes $f_1$ instead. Our example illustrates that a given test case for a configurable system can have different coverage with different configurations.

| **Configuration $C$** | **Configuration $C'$** |
|---|---|
| $P_1$: OFF | $P_1$: ON |
| $P_2$: OFF | $P_2$: OFF |

Figure 1 Pre-selected configurations for testing $S$

### B. Notations

Let $T=\{t_1, t_2, ..., t_n\}$ be an existing test suite with $n$ test cases used for testing a configurable system $S$ with $m$ configurable options. Let $C=\{P_1^C, P_2^C, ..., P_m^C\}$ and $C'=\{P_1^{C'}, P_2^{C'}, ..., P_m^{C'}\}$ be two configurations selected for testing $S$.

Let $\Delta(C, C')$ denote the set of options whose values are different between $C$ and $C'$ (henceforth, we use $\Delta$ to refer to $\Delta(C, C')$). By definition, $\Delta = \{P_i \mid P_i^C \neq P_i^{C'}, i \in [1,m]\}$.

A given configuration option, $P_i$, can be mapped to its corresponding variables in the source code (see Section IV.B for details). Let $imp(P_i)$ represent the code in $S$ that is statically impacted by the variables corresponding to $P_i$. Let $imp(\Delta)$ represent the code that is statically impacted by the variables corresponding to options in $\Delta$. Specifically, let $imp^f$ ($imp^s$) be a set of functions (statements) that are impacted. Let $cov(C, t_i)$ represent the code that $t_i$ executes when $S$ is tested under $C$. Specifically, let $cov^f(C,\ t_i)$ ($cov^s(C,\ t_i)$) be the functions (statements) that $t_i$ covers.

### C. Problem Definition

If $S$ is tested with test suite $T$ under $C$, the problem addressed by our approach is to select $T' \subseteq T$, to test $S$ under $C'$. In other words, the problem is to determine for each $t_i \in T$, if $t_i$ belongs to $T'$.

```
S(int x){
  #if P₁              // P₁ = ON
    if (x == 0)       // test case t₀ executes
        f₁();
    if(x == 1)        // test case t₁ executes
        f₃(-29);
  #else               // P₁ = OFF
      if (x == 0)
          f₂();
      if(x == 1)
          f₃(25);
  #endif
}

void f₁(){
  printf("Executed only when P₁ is ON.");
}

void f₂(){
  printf("Executed only when P₁ is OFF.");
}

void f₃(int x){
   printf("This  statement is not controlled
           by P₁.");
   if(x < 0)
      printf("This statement is executed
             only when P₁ is ON.");
}
```

```
Test case t₀: S(0)
Test case t₁: S(1)
```

Figure 2 An example configurable system with two test cases

### D. Overview

The essential goal of testing $S$ under a new configuration $C'$ is to execute the code in $S$ that has not been covered when $S$ is tested under the old configuration $C$. To achieve this goal, a test case should be selected if it covers different code when executed under $C'$ that is not covered by the same test case when executed under $C$. Formally, $t_i$ should be selected if the set difference, $cov(C', t_i) - cov(C, t_i) \neq \emptyset$. However, the dynamic $cov(C', t_i)$ is unavailable in practice unless we rerun the full test suite $T$ under $C'$. Given that $t_i$ and the source code of $S$ are unchanged, the different coverage of $t_i$ in $S$ under $C'$ can only be caused by the configuration changes. Hence, our approach conservatively approximates the dynamic coverage difference (i.e., $cov(C', t_i) - cov(C, t_i)$), which is not available in practice, using static impact of configuration changes. Accordingly, our approach selects $t_i$ for testing $S$ under configuration $C'$ if $t_i$ covers, under configuration $C$, parts of the source code that are impacted by the configuration changes ($\Delta$). Formally, our approach selects $t_i$ if $cov(C, t_i) \cap imp(\Delta) \neq \emptyset$. Code coverage can be measured at different granularities, such as statement, block, or function. Our approach uses function-level granularity because a function is typically considered as a unit in testing. Hence, our approach selects $t_i$ for testing $S$ under $C'$ if

$$cov^f(C, t_i) \cap imp^f(\Delta) \neq \emptyset \qquad (1)$$

## E. Details

Our approach involves four steps as shown in Figure 3: (1) computing configuration differences between $C$ and $C'$, (2) computing the functions impacted in $S$ by the configuration differences, (3) computing the function coverage for each test case $t_i$ in $T$ when $S$ is tested under $C$ dynamically, and (4) selecting $t_i$ based on Equation (1).

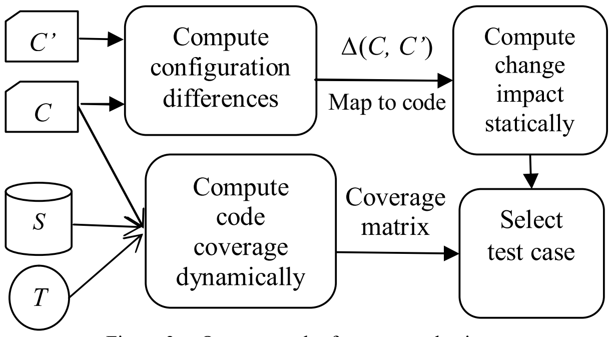
C'
C
S
T
Compute configuration differences
Δ(C, C')
Map to code
Compute change impact statically
Compute code coverage dynamically
Coverage matrix
Select test case

Figure 3 Our approach of test case selection

Next, we describe these steps with the example configurable system $S$ shown in Figure 2.

1) *Computing Configuration Differences ($\Delta$)*
For each option $P_i$ ($i \in [1, m]$), our approach compares $P_i^{C'}$ with $P_i^{C}$: if they are different, $P_i$ is included in set $\Delta$. In our example, $m$=2 and $\Delta = \{P_1\}$.

2) *Computing Configuration Change Impact ($imp^f(\Delta)$)*
For each $P_i \in \Delta$, our approach maps $P_i$ to its corresponding variable(s) in the source code (see Section IV.B for details) and then computes $imp^f(P_i)$. By combining all impact sets,

$$imp^f(\Delta) = \cup_{P_i \in \Delta} imp^f(P_i) \qquad (2)$$

In our example, $imp^f(\Delta) = imp^f(\{P_1\}) = \{f_1, f_2, f_3\}$, since $P_1$ impacts $f_1$, $f_2$, and $f_3$.

3) *Computing Code Coverage*
While executing $T$ under $C$, our approach computes $cov^f(C, t_i)$ for each $t_i \in T$. The left half of TABLE I shows the resulting function coverage matrix. Value 1 (0) represents that a function is covered (not covered) by a test case under a specific configuration. In our example, $cov^f(C, t_0) = \{f_2\}$ and $cov^f(C, t_1) = \{f_3\}$.

4) *Selecting Test Cases*
According to Equation (1), our approach selects $t_0$ since $cov^f(C, t_0) \cap imp^f(\Delta) = \{f_2\} \neq \emptyset$. For the same reason, $t_1$ is also selected.

TABLE I FUNCTION COVERAGE MATRICES UNDER *C* AND *C'*

| | ***C*** | | ***C'*** | |
|---|---|---|---|---|
| | ***$t_0$*** | ***$t_1$*** | ***$t_0$*** | ***$t_1$*** |
| ***$f_1$*** | 0 | 0 | 1 | 0 |
| ***$f_2$*** | 1 | 0 | 0 | 0 |
| ***$f_3$*** | 0 | 1 | 0 | 1 |

TABLE II STATEMENT COVERAGE MATRICES UNDER *C* AND *C'*

| | ***C*** | | ***C'*** | |
|---|---|---|---|---|
| | ***$t_0$*** | ***$t_1$*** | ***$t_0$*** | ***$t_1$*** |
| ***$f_1$*** | 0 | **0** | 16 | 0 |
| ***$f_2$*** | 20 | 0 | 0 | 0 |
| ***$f_3$*** | 0 | **24** | 0 | 24,25,26 |

## F. Safety Criterion of Our Approach

Our test case selection approach is safe if it selects all test cases in $T$ that have different coverage in $S$ under $C$ and $C'$. But the dynamic coverage difference for $t_i$, i.e., $(cov(C, t_i) - cov(C', t_i)) \cup (cov(C', t_i) - cov(C, t_i))$, which we denote by $< cov(C, t_i) - cov(C', t_i) >$, is unavailable in practice. Hence, as discussed in Section D, a full recall of the dynamic coverage difference by the static configuration change impact is a necessary and sufficient condition for the safety of our approach. Specifically, the *safety criterion* can be described as follows. For each $t_i$, the code which is statically impacted by configuration changes must contain the code that is covered by $t_i$ under $C'$ but not by $t_i$ under $C$ and vice versa. Formally,

$$imp(\Delta) \supseteq < cov(C, t_i) - cov(C', t_i) >, \ t_i \in T \qquad (3)$$

or

$$imp(\Delta) \supseteq \cup_{t_i \in T} < cov(C, t_i) - cov(C', t_i) > \qquad (4)$$

For the LHS of Equation (4), $imp(\Delta)$ is computed with Equation (2) at the function granularity. Next, we discuss how the granularity used for measuring coverage affects the RHS of Equation (4), and hence, the accuracy of the safety criterion.

First, we consider the case where coverage is measured at the function granularity. The function coverage matrices for $S$ under $C$ and $C'$ with $T$=$\{t_0, t_1\}$ are shown in TABLE I. From this table, $< cov^f(C, t_0) - cov^f(C', t_0) > = \{f_1, f_2\}$ and $< cov^f(C, t_1) - cov^f(C', t_1) > = \emptyset$. According to Equation (4), the safety criterion is $imp^f(\Delta) \supseteq \{f_1, f_2\}$. However, this criterion is not accurate since $f_3$ should also be included in the RHS of Equation (4). In our example, $t_1$ covers $f_3$ under both $C$ and $C'$, but at the statement level, it executes more statements in $f_3$ when tested under $C'$. This difference is not captured when the coverage is measured at the function granularity.

Next, we consider the case where coverage is measured at the statement granularity. The statement coverage matrices for $S$ under $C$ and $C'$ with $T=\{t_0, t_1\}$ are shown in TABLE II. Each entry represents the line numbers covered by a particular test case in a particular function. The bold font numbers indicate that $t_1$ covers no statements of $f_1$ but covers statement 24 of $f_3$. From this table, $< cov^s(C, t_0) - cov^s(C', t_0) > = \{16,20\}$, which belongs to functions $f_1$ and $f_2$; and $< cov^s(C, t_1) - cov^s(C', t_1) > = \{25,26\}$, which belongs to $f_3$. According to Equation (4), the safety criterion is $imp(\Delta) \supseteq \{f_1, f_2, f_3\}$, which accurately captures the coverage differences. In Section IV, we present an empirical study that verifies the safety of our approach.

Although coverage measured at the statement-level granularity is appropriate for the safety analysis of our test selection approach, our approach measures coverage at the function-level as defined in Equation (1), because coverage measured at finer granularities (e.g., statement-level) incurs more cost, while coverage measured at coarser granularities (e.g., file-level) leads to imprecision, and hence, too many false positives in test selection. A function-level granularity makes a trade-off between cost and precision.

### *G. Static Analysis Used by Our Approach*

Theoretically, we expect that the impact set generated with expensive static analysis settings should have better recall of dynamic coverage differences compared to the impact set generated with less expensive static analysis settings. As discussed in Section F, the recall of the dynamic coverage differences by the static impact decides the safety of our approach. Hence, by default, our approach uses the most expensive static analysis settings available. However, the time overhead may be prohibitive with expensive static analysis settings, especially for larger systems. In Section IV, we present a study comparing the impact sets generated with different static analysis settings.

### *H. Computing Impact of Multiple Option Changes*

Besides considering less expensive static analysis settings to lower the cost of our approach, we also explore other opportunities to save the time incurred by the static analysis. Let $\Delta=\{P_{ch1}, \ldots, P_{chk}\} \subseteq P$ $(1 \leq k \leq m)$, where $P_{ch1} \ldots P_{chk}$ denote $k$ options that change between $C$ and $C'$. When the change involves multiple option changes ($k > 1$), we compute $imp(\Delta)$ as the union of impact of each option in $\Delta$ (Equation (2)). We call this the *union* method. Alternatively, we may merge variables corresponding to each option in $\Delta$ as one single input to static analysis. Our approach then computes the impact of all changes at once. We call this the *multi-select* method. Compared to the union method, the multi-select method may largely reduce the redundant overhead of rerunning static analysis for each option change. The union method invokes the analysis multiple times but the multi-select method invokes the analysis only once. However, the multi-select method may be used only if it is a conservative approximation of the union method for computing the impact, formally,

$$imp(P_{ch1} + \cdots + P_{chk}) \supseteq \cup_{i=1}^{k} imp(P_{chi}).$$

In Section IV, we present a study comparing the impact sets generated by the *union* method and *multi-select* method.

## III. IMPLEMENTATION

Our approach employs a static analysis tool, *CodeSurfer* [28], for computing configuration change impact. *CodeSurfer* uses forward program slicing to compute $imp(P_i)$. Forward slicing, required for computing the impact sets, depends on several static analysis parameters such as pointer analysis, context/flow sensitivity, non-local analysis, and slicing dependency analysis.

TABLE III STATIC ANALYSIS SETTINGS USED IN OUR EXPERIMENTS

| | ***H*** | ***L*** | ***H_cd*** | ***H_dd*** | ***L_cd*** | ***L_dd*** |
|---|---|---|---|---|---|---|
| *non-locals* | *yes* | *no* | *yes* | *yes* | *yes* | *yes* |
| *dependencies* | *both* | *both* | *cd* | *dd* | *cd* | *dd* |

We did several experiments to determine that the static analysis parameters non-local analysis and slicing dependency analysis had the most significant impact on the time and safety of our test selection approach. Hence, in our experiments, we do not consider other static analysis parameters and conservatively use the most expensive value available for them (for example, for pointer analysis, we always uses the Andersen's algorithm [1]).

The non-locals of a function include all the global variables and indirectly accessed variables used or modified by a function. With *CodeSurfer*, the non-local analysis (in short, *non-locals*) is configurable and can be shut off. There are three options for slicing dependency analysis (in short, *dependencies*): control-dependence only (*cd*), data-dependence only (*dd*), and both (*both*). Depending on the option, the forward slicing follows control-dependence edges only, data-dependence edges only, or both edges. With two choices for *non-locals* (*yes/no*) and three for *dependencies* (*cd/dd/both*), there are six static analysis settings (set of static analysis parameters) altogether, as shown in TABLE III (*H* and *L* represent *H_both* and *L_both* respectively). These different settings decide the expense of our static analysis. For example, setting *H* is the most expensive, while *L_cd* and *L_dd* are the least expensive.

## IV. STUDY

In this section, we present a study to examine the safety of our test case selection approach on *make* [30]. We also present a study on a large industrial system developed at ABB, to explore several opportunities to reduce the time taken by the static analysis used by our approach. We have the following research questions:

- RQ1. Is our approach safe for test case selection when using the most expensive static analysis settings?
- RQ2. What percentage of functions is impacted by a configurable option change? Hence, what percentage of testing cost can our approach potentially save?

- RQ3. How do the impact sets generated with less expensive static analysis settings compare with those generated by the most expensive static analysis settings?
- RQ4. How do the impact sets computed by the *multi-select* and the *union* method compare?

## A. *Study Subjects*

We study the safety of our approach on *make* [30], an open source program written in C. *make* is a widely popular and general purpose tool used to compile programs. The source code for *make* (version 3.78.1) is available at the GNU website [30]. There are about 15,000 uncommented lines of code in the source code of this version.

We also apply our approach to a core component (called *ABB1* hereafter) of a large real-time embedded software system developed at ABB. *ABB1* consists of 1.18 MLOC written in C/C++. It contains 20,432 functions across 58 modules. Each module defines a subsystem that implements different functionalities of the system.

In some configurable systems such as *make* and *vim*, each configurable option is mapped to a unique global variable, which is used throughout the system. Different from these systems, *ABB1* stores the values of its configurable options in a database. When an option is to be used, its value is retrieved from the database and assigned to a locally defined variable. Therefore, a configuration option in *ABB1* may map to multiple variables in the system. Specifically, there are 129 configurable options used in *ABB1*, distributed among 394 variables. Some options map to only one variable while others map to multiple variables ranging from 2 to 37. Code coverage data is unavailable for *ABB1* (we discuss this issue in Section VI.B).

## B. *Study of the Safety of Our Approach*

By default, our approach uses the most expensive static analysis settings available with *CodeSufer*. Based on Equation (4), we examine the safety of our approach by comparing $imp^f(\Delta)$ with $< cov^s(C, t_i) - cov^s(C', t_i) >$. The whole process of our safety study is broken into multiple steps: (1) creating configurations and test cases, (2) executing test cases under different configurations, (3) computing change impact, $imp^f(\Delta)$, with static analysis, (4) computing code coverage differences, $< cov^s(C, t_i) - cov^s(C', t_i) >$, and (5) comparing change impact with code coverage differences. Each step is described next.

1) *Creating configurations and test cases*

The test suite for *make* was obtained from the Software Infrastructure Repository (SIR) [7]. This test suite has been previously studied in a related research [16]. We modified the test suite by separating out test cases with configurable options, leaving behind 540 test cases in the test suite. We also select 12 configurable options and create seven configuration instances using a commonly used sampling approach [6][17].

2) *Executing test cases*

We run 540 test cases under each configuration on a GNU/Linux machine. We instrument the code using *gcov* [30], run the tests, and record the statement coverage data as defined in Section II.F and shown in TABLE II. Recall that though our test case selection approach measures coverage at the function granularity, for our study of safety, the coverage is measured at the statement granularity (see Section II.F).

```
1: struct command_switch
2: {
3:   int c;      /* The switch character.  */
4:   enum        /* Type of the value.  */
5:   {
6:      flag,
7:      flag_off,
8:      string,
9:      positive_int,
10:     floating,
11:     ignore
12:   } type;
13:
14:   char *value_ptr;  /* Pointer to the
       value-holding variable.  */
15:
16:   unsigned int env:1;
17:   unsigned int toenv:1;
18:   unsigned int no_makefile:1;
19:
20:   char *noarg_value; /* Pointer to value
       used if no argument is given.  */
21:   char *default_value; /* Pointer to
       default value.  */
22:
23:
24:   char *long_name; /*Long option name. */
25:   char *argdesc;
26:   char *description;
27: };
28:
29:
30:/* The table of command switches.  */
31: static const struct command_switch
    switches[] =
32: {
        … …
50:   {'s', flag, (char *) &silent_flag, 1, 1,
     0, 0, 0, "silent", 0, ("Don't echo
     commands")},
          … …
   }
```

Figure 4 Data structure and values of options for *make*

3) *Computing configuration change impact*

There are seven configurations, resulting in 21 pairs of different configurations. For each pair, we first identify the differences (i.e., Δ) between configurations in that pair. Next, we map the options in Δ to their corresponding variables in the source code. Our mapping process for *make* is illustrated as follows.

Figure 4 shows the `main.c` file of *make* (some comments and lines are omitted for clarity). Lines 1 to 27 define a data structure, `command_switch`, which is used for holding configuration option information for *make*. The first member in the data structure, `c`, holds the character (symbol) of the option (Line 3). The third member, `value_ptr`, is used to hold the value of the option (Line 14). Rest of the code from Line 30 onwards defines a concrete object of this structure, `switches`, which is an array containing multiple options. Each item in the `switch` array represents an option. For example, `s` is an option (defined in Line 50) that determines if the various compile commands are echoed when *make* compiles a program. The value of this option is held in the variable `silent_flag`. Hence, we regard `silent_flag` as the corresponding variable in the source code for option `s`. Following the same procedure, every option in Δ is mapped to its corresponding variable in the source code. By running *CodeSurfer* with *H* setting on each option in Δ, we get $imp^f(\Delta)$ (Equation (2)).

4) *Computing coverage difference*

For each configuration, there is a corresponding statement coverage matrix (see TABLE II for an example) generated from *gcov*. For each pair of configurations, we compute $< cov^s(C, t_i) - cov^s(C', t_i) >$ for each test case $t_i$.

5) *Safety of our approach*

For each pair of configurations, we compare $imp^f(\Delta)$ with the union of all $< cov^s(C, t_i) - cov^s(C', t_i) >$ where $i \in [1, 540]$.

## C. Study on ABB1

For the research questions related to the study of *ABB1*, RQ2-RQ4, the dependent variable is measured as the percentage of the functions impacted in *ABB1*.

In *ABB1*, there are 129 configurable options distributed among 394 variables. Our static analysis module takes each variable as an input and analyzes it with six settings (defined in Section III): *H*, *H_cd*, *H_dd*, *L*, *L_cd*, and *L_dd*. These settings come from the combination of two independent variables: $I_1$ (*H*, *L*) and $I_2$ (*cd*, *dd*, *both*). *CodeSurfer* generates 2364 (394×6) impact sets in total. Each impact set consists of the impacted lines, impacted functions, and the number of impacted lines and functions.

For RQ2, we calculate the percentage of the functions that is impacted by a single configurable option change, with the *H* setting. We group the impact sets by configurable options. For each option, we count the total number of impacted functions across all variables and calculate the percentage over all 20432 functions in *ABB1*.

For RQ3, our goal is to compare the impact sets generated with less expensive static analysis settings and the most expensive static analysis setting possible. As described in the beginning of this section, there are six impact sets generated by *CodeSurfer* corresponding to six setting combinations of $I_1$ and $I_2$. For each impact set, we calculate the percentage of impacted functions for each variable. We compare these impact sets with respect to $I_1$ and $I_2$.

For RQ4, the independent variable is the method of calculating impact of multiple option changes, which can either be the *union* method or the *multi-select* method (defined in Section II.H). Particularly, we examine configuration changes that involve only two changed options between *C* and *C'*. We study changes involving only a pair of option changes as any changes with three or more option changes can be seen as multiple pairs of option changes. With 129 options in *ABB1*, there are over 8000 pairs of options. Since the computation with *union* or *multi-select* method is independent of the options being computed, it suffices to study a random selection of option pairs. Hence, we randomly select 10 pairs, $(P_1^i, P_2^i)$ $(i \in [1,10])$. For each pair *i*, let $\{a_{1i}, a_{2i}, \dots a_{pi}\}$ and $\{b_{1i}, b_{2i}, \dots b_{qi}\}$ represent variables mapped from $P_1^i$ and $P_2^i$, respectively. The numbers of the variables for each option (denoted as *p* and *q*) in a pair are shown in TABLE IV. For each pair, *CodeSurfer* generates $imp(a_{1i}+a_{2i}+\dots+a_{pi}+b_{1i}+b_{2i}+\dots+b_{qi})$ by the *multi-select* method and $(\bigcup_{k=1}^{p} imp(a_{ki})) \cup (\bigcup_{k=1}^{q} imp(b_{ki}))$ by the *union* method. By default, *CodeSurfer* uses the most expensive *H* setting for both methods.

TABLE IV NUMBER OF VARIABLES COREESPONDING TO EACH OPTION IN THE SELECTED OPTION PAIRS

| **pair *i*** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
|---|---|---|---|---|---|---|---|---|---|---|
| ***p*** | 1 | 1 | 2 | 1 | 1 | 9 | 1 | 1 | 1 | 5 |
| ***q*** | 1 | 1 | 1 | 2 | 6 | 1 | 6 | 1 | 2 | 1 |

## V. RESULTS

In this section, we present the results of our study with respect to each research question, presented in Section IV. All *impact* is measured in terms of the *percentage of impacted functions*.

## A. RQ1: Safety of Our Approach

For RHS of Equation (4), our approach runs the test suite *T* (540 test cases) under each configuration, with *gcov* to collect statement coverage, $cov^s(T,C_i)$, $i \in [1,7]$. For each pair of configurations *C* and *C'* $(C, C' \in \{C_i\}, i \in [1,7]$ and $C \neq C')$, we compute the functions whose code coverage is different in *C* and *C'*, are shown in TABLE V. The number ranges from 0 to 101. As for the LHS of Equation (4), the number of impacted functions computed by our static analysis ($imp^f(\Delta)$) is 294 for all pairs.

TABLE V NUMBER OF FUNCTIONS OF DIFFERENT COVERAGE

| | | | | | |
|---|---|---|---|---|---|
| $C_1, C_2$ | 91 | $C_2, C_4$ | 82 | $C_3, C_7$ | 97 |
| $C_1, C_3$ | 99 | $C_2, C_5$ | 9 | $C_4, C_5$ | 90 |
| $C_1, C_4$ | 43 | $C_2, C_6$ | 10 | $C_4, C_6$ | 90 |
| $C_1, C_5$ | 99 | $C_2, C_7$ | 101 | $C_4, C_7$ | 55 |
| $C_1, C_6$ | 99 | $C_3, C_4$ | 90 | $C_5, C_6$ | 1 |
| $C_1, C_7$ | 23 | $C_3, C_5$ | 0 | $C_5, C_7$ | 97 |
| $C_2, C_3$ | 9 | $C_3, C_6$ | 1 | $C_6, C_7$ | 97 |

We compare the sets of functions whose code coverage is different in *C* and *C'*, with the impact sets. In 9 among 21 cases, $imp^f(\Delta) \supseteq \bigcup_{t_i \in T} < cov^s(C, t_i) - cov^s(C', t_i) >$, that is, satisfying the safety criterion. Other cases all share the same exception: there are only two functions that belong to $\bigcup_{t_i \in T} < cov^s(C,t_i) - cov^s(C',t_i) >$ but not in *imp*$^f$(Δ). The first one is the function `perror_with_name`. However, `perror_with_name` calls function `error`, which is contained in *imp*$^f$(Δ). Our approach will select all test cases that cover `error`, and the test cases covering `perror_with_name` are only a subset of them. In other words, our approach will not miss any test case that covers `perror_with_name`. The second function is the function `fatal`. Let *F* be a set of functions that call `fatal`, then `fatal` can only be covered by test cases that cover functions in *F*. In our study, all functions that call `fatal` are contained in *imp*$^f$(Δ). Our approach will select all test cases that cover functions in *F*, which contains all test cases that may cover `fatal`. Hence, the fact that functions `perror_with_name` and `fatal` are not contained in *imp*$^f$(Δ) does not affect the test case selection safety of our approach. In summary, we have empirically confirmed that the impacted functions computed by our static analysis are a conservative approximation of coverage differences, and hence our test case selection approach is expected to be safe.

Moreover, the set of impacted functions computed statically is larger than the set of functions in the dynamic coverage difference. This may be because that our static analysis considers both control and data flow, while the dynamic coverage data only considers the control flow.

## B. *RQ2: Impact of Configuration Changes*

To address RQ2, we examine the impact of changing a single configurable option. Figure 5 shows a box plot for the change impact for all 129 options. Changes to most of the options only impact less than 10% of all the functions in *ABB1*. The largest impact by an option change is no more than 20% of all the functions. In *ABB1*, the test cases are evenly distributed among the units (functions) that are tested. In other words, each function under test is tested by almost the same number of test cases. Given this even distribution of test cases, our results indicate that only about 20% of the test cases may have to be selected for *ABB1* if a configurable option changes. Considering that large systems usually have hundreds or even thousands of configurations to test, our approach will lead to substantial savings.

## C. *RQ3: Impact Sets and the Expense of Static Analysis*

In RQ3, we examine whether our approach computes significantly smaller impact with less expensive static analysis settings (i.e., less expensive settings of $I_1$ and $I_2$), compared to that with the highest setting. TABLE VI shows the statistics for the impact computed by our approach with six different settings. At a first glance, it appears that the impact sets computed by *CodeSurfer* with the less expensive settings are much smaller than that with the higher setting. For example, the average (mean) impact is 3.318% with *H*, and only 0.686%, 0.005%, and 0.046% with *L*, *H_cd*, and *H_dd* respectively.

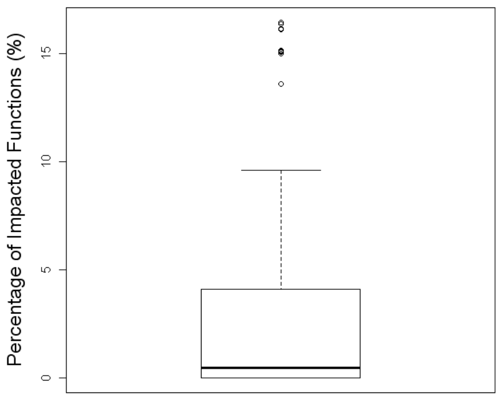

Figure 5 Impact computed with setting *H* (per option)

We examine if the differences in the values in different impact sets are statistically significant. In other words, we examine if setting $I_1$ and/or $I_2$ has statistically significant effect on the impact results. We apply a *two-factor ANOVA test* [12] on all our impact results. TABLE VII shows our results. The first column lists each individual setting followed by the interaction between the settings denoted by "$I_1 \times I_2$". The last column shows the probability of the null hypothesis: *there is no significant effect of this setting (or of the interaction between settings).* The probability of $I_1 \times I_2$ is smaller than $2.2\times10^{-16}$ (shown in bold font), which is smaller than a typical $\alpha$ level of 0.05. This indicates that we should reject the null hypothesis and accept the alternative hypothesis: *there is a significant interaction between $I_1$ and $I_2$*. Due to this significant interaction, we are unable to examine the *main effects* of each setting; instead, we examine their *simple effects* [12].

TABLE VI IMPACT (%) COMPUTED WITH VARIOUS SETTINGS

| | *L* | *L_cd* | *L_dd* | *H* | *H_cd* | *H_dd* |
|---|---|---|---|---|---|---|
| max | 13.590 | 0.005 | 0.049 | 16.450 | 0.005 | 8.374 |
| 3rd qtr. | 0.661 | 0.005 | 0.020 | 4.126 | 0.005 | 0.020 |
| **mean** | **0.686** | **0.005** | **0.014** | **3.318** | **0.005** | **0.046** |
| median | 0.230 | 0.005 | 0.015 | 0.499 | 0.005 | 0.015 |
| 1st qtr. | 0.015 | 0.005 | 0.005 | 0.025 | 0.005 | 0.005 |
| min | 0.005 | 0.005 | 0.005 | 0.005 | 0.005 | 0.005 |

TABLE VII ANOVA TEST RESULTS FOR RQ2

| | Degree of Freedom | F value | Probability (>F) |
|---|---|---|---|
| $I_1$ | 1 | 94.76 | $< 2.2\times10^{-16}$ |
| $I_2$ | 2 | 210.52 | $< 2.2\times10^{-16}$ |
| $I_1 \times I_2$ | 2 | 91.45 | $\mathbf{< 2.2\times10^{-16}}$ |

1) *Simple Effects of $I_1$ : H vs. L*

First, we examine the effect of $I_1$. Figure 6 shows the box plots comparing the impact computed with different settings of $I_1$ (*H* and *L*), for all variables, grouped by $I_2$. The figure shows that for the "*both*" setting of $I_2$ (left box plots), impact computed with the *H* setting is a superset of the impact computed with the *L* setting. For the "*cd*" and "*dd*" settings of $I_2$ (middle and right box plots), the impact computed with *H* and *L* are similar. We also apply a *t-test* [12] on the impact set of the first group (i.e., "*both*" setting of $I_2$). The result indicates that the difference is statistically significant. Based on this examination, if we choose setting "*both*" for $I_2$, *CodeSurfer* with less expensive settings of $I_1$ (i.e., *L*) will compute impact sets that are significantly smaller.

2) *Simple Effects of $I_2$: both vs. cd vs. dd*

Next, we examine the effect of $I_2$. Figure 7 shows the box plots comparing the impact computed with different settings of $I_2$ (*cd*, *dd*, and *both*), for all variables, grouped by $I_1$. Our results show that impact sets with "*both*" are significantly larger than those with "*cd*" and "*dd*", both for *H* (left box plots) and *L* (right box plots). We also apply a t-test on the impact sets of both groups. The results are consistent with what we observed from the figure. Based on our study, less expensive setting of $I_2$ (i.e., *cd* or *dd*) will compute impact sets that are significantly smaller.

3) *Summary*

From our analysis of both $I_1$ and $I_2$, to compute the impact of configuration changes for *ABB1*, less expensive static analysis settings result in significantly smaller impact. As a result, it appears that from the perspective of test selection safety, it is better for *CodeSurfer* to be run with *H* setting, since the settings of both $I_1$ and $I_2$ can result in significantly different impact sets. However, if the resource constraints only allow *CodeSurfer* to be run with a less expensive setting, among all other five choices (*H_cd, H_dd, L, L_cd, and L_dd*), *L* will result in impact that has better recall of the coverage differences. This indicates that considering both control and data dependencies may be more important than nonlocal analysis for the safety of test case selection when using less expensive settings.

## D. *RQ4: Multi-select vs. Union Method*

TABLE VIII shows the number of impacted functions computed by the *union* and the *multi-select* methods for the 10 pairs of configurable options. While seven impact sets are identical, for the remaining three, impact sets computed by the *union* method are smaller compared to that computed by the *multi-select* method.

TABLE VIII NUMBER OF IMPACTED FUNCTIONS COMPUTED BY DIFFERENT METHODS

| pair | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Union | 97 | **2199** | **2108** | 6 | 3063 |
| Multi-select | 97 | **2284** | **2193** | 6 | 3063 |
| pair | 6 | 7 | 8 | 9 | 10 |
| Union | 297 | 3060 | **2108** | 3060 | 274 |
| Multi-select | 297 | 3060 | **2193** | 3060 | 274 |

Our results also show that for each pair, the impacted functions computed by the *union* method is a subset of the impacted functions computed by the *multi-select* method. This indicates that it is safe to calculate the impact of multiple option changes with the *multi-select* method instead of the *union* method. The *multi-select* method incurs lesser time overhead with no significant loss in safety.

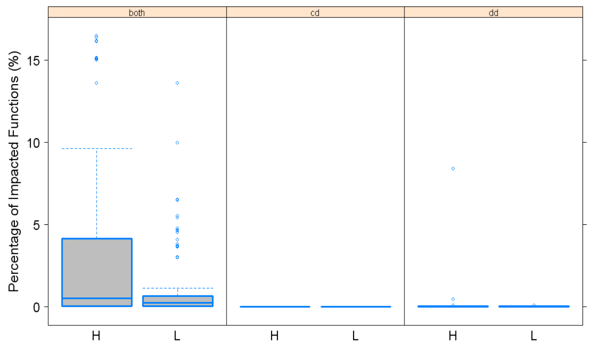

Figure 6 Impact by different setting of $I_1$ (grouped by $I_2$)

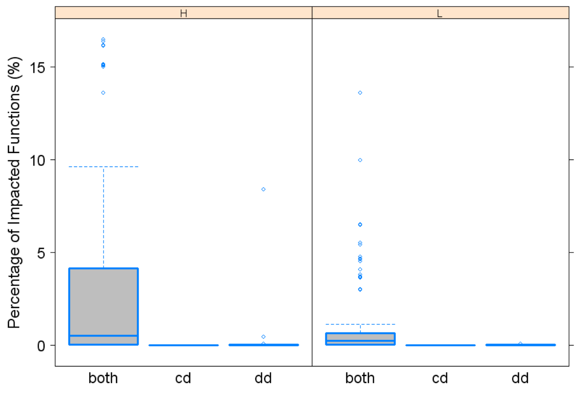

Figure 7 Impact by different setting of $I_2$ (grouped by $I_1$)

## VI. DISCUSSION

In this section, we discuss the limitations and threats to validity related to our approach.

## A. *Configuration Mappings*

We have examined the safety of our approach with an empirical study on *make*. However, we should notice that the safety is also dependent on the correctness and completeness of the mappings between configurable options and their corresponding variables in the source code. In *make*, the mapping of configurable options to variables in source code is straightforward. However, in complex systems, it is often difficult to accurately map configurable options to the variables in the source code. The accurate mapping requires a thorough understanding of the system specifications and a good communication with the system developers. The configuration of *ABB1* is complex and the mappings between each option and its variables were manually derived. We have carefully read through the configuration specification and the comments in source code that describe how configuration options are defined, loaded, and used in the system. But we might have still missed some options and their corresponding variables.

## B. *Applicability and Generality*

The study of different static analysis settings help us decide if we can save cost by choosing less expensive static setting without losing significant impact. However, the correlation between the static analysis expense and the size of impact set is system dependent, because different systems define and implement configurations differently. Therefore, our results may not generalize to other systems. Furthermore, though the static analysis with lower settings of both $I_1$ and $I_2$ result in significantly smaller impact sets compared to that with the highest setting, it does not necessarily mean that the selections based on lower settings are unsafe. We will investigate the interplay between static analysis and safety as a part of our future work.

### 1) *Test Case Selection without Coverage Data*

For some systems, the function coverage data ($cov^f(T,C)$) might not be available. For such systems, either the testing was done without the coverage tool or the coverage tool was not compatible with the third-party applications used by the system. In such cases, some approximations may be used for the function coverage data. For example, in *ABB1*, test cases were created for testing different functionalities of the system. Hence, we can map each test case to the functionalities that it tests. Furthermore, particular system functionality is implemented in one or more functions. Hence, we can also map the functionality to its functions. Given the mappings between test cases and functionalities, and between functionalities and functions, we can approximate the set of functions that are "covered" by a particular test case even when the actual function coverage information is unavailable.

## VII. RELATED WORK

In this section, we introduce related literature on test case selection, configurable system testing, and impact analysis of configuration changes.

### A. *Test Case Selection*

In traditional regression testing, given an initial version of a system *S* and a test suite *T*, a subset of tests *T'* has to be selected from *T* to test a new version *S'* (due to source code changes) of *S*. *Ad-hoc* selection approaches [10] are not safe because they may miss some test cases that reveal faults in the modified program. *Safe* test case selection approaches [21] select *all* test cases in the original test suite, which can reveal faults in the modified program. Rothermel et al. present a safe regression test selection approach [22] based on analyzing the Control Flow Graphs (CFG) of *S* and *S'*.

Our approach is complementary to traditional regression testing approaches. While regression testing approaches select test cases between multiple versions of a system (when the code changes), our approach selects test cases for testing a single version of a system (no code changes) under different configurations.

### B. *Testing Configurable Systems*

As introduced in Section I, configuration selection (prioritization) and test case selection for the selected configurations are two different dimensions of testing configurable systems. To our best knowledge, existing approaches [5][17][19] only deal with the first dimension, i.e., configuration selection and prioritization. In contrast, our approach deals with the second dimension, i.e., test case selection for the pre-selected configurations.

*Combinatorial Interaction Testing* (CIT) [4] is commonly used for sampling configurations. Many studies [6][11] examine the effectiveness of CIT to model configurations and the impact of configurations on testing. Moreover, Reisner et al. [19] show that most code can be covered if a system is tested by sampling its configurations by considering lower levels of interactions between options.

Approaches for regression testing of configurable systems [9][26][27] study the issue of improving the effectiveness of configuration sampling for testing a new version of system. Qu et al. [17] study the impact of configurations across multiple versions of a system and prioritize configurations to improve early fault detection.

Product lines are commonly regarded as configurable systems. Scheidemann [24] proposed an approach to select a small set of vehicle configurations for product verification, such that the successful verification of this small set implies the correctness of the entire product family.

However, none the aforementioned approaches deal with selecting test cases for testing a system under sampled or prioritized configurations.

Recently, Nanda et al. [14] proposed an approach to support regression testing of systems that exhibit frequent changes in non-code parts of the system. They focus on changes to databases and simple configuration files. However, their approach focuses on a simple model of property files, and hence, cannot handle complex configurations. In comparison, our approach handles different types of configurations by mapping configurable options to the corresponding variables in the source code. Moreover, as our approach analyzes the configuration changes on source code through conservative static program slicing, our impact results can lead to a safe test selection.

### C. *Impact Analysis of Configuration Changes*

Dor et al. [8] investigate the impact of configuration changes in an *ERP* system through program slicing [24]. The impact information is then used by the system's customers and maintainers to understand the change and select the right values for the configuration options. However, their approach does not handle test case selection. Also, their impact analysis tool was specifically designed for ERP systems, and hence, their approach is not generically applicable to other types of systems. In contrast, our approach is not limited to any particular type of systems.

Robinson and White [18] propose an approach and develop a tool, *Firewall*, to analyze the impact of configuration changes for selecting test cases. Their approach only identifies local impact which is one calling level away from the change. But in fact, as evidenced by our empirical study (presented in Section V.C), configuration impact usually propagates globally beyond one calling level, across different functions, files, and modules. Therefore, their test case selection based on local impact is not safe.

## VIII. CONCLUSIONS

In this paper, we proposed an approach to select test cases when configuration under test changes. Our approach uses the dynamic coverage data of the test suite under previously tested configuration and the static impact of configuration differences for selecting test cases. Our results demonstrate that our approach is safe. Our results also indicate that only about 20% of test cases need to be selected for *ABB1* when a configuration under test changes, thereby reducing the testing cost. As a part of future work, we will study the different static analysis settings [2] and test-selection safety tradeoffs that might be possible with our approach. We will also explore how configuration selection and test case selection for the selected configurations can be combined.

## ACKNOWLEDGMENTS

We thank GrammaTech Inc. for providing timely technical support for *CodeSurfer*.

## REFERENCES

[1] L. O. Andersen. Program analysis and specialization for the C programming language. PhD thesis. DIKU University of Copenhagen, 1994.

[2] M. Acharya and B. Robinson. Practical change impact analysis based on static program slicing for industrial software systems. *International Conference on Software Engineering (ICSE)*, 2011, pp. 746-755.

[3] R. S. Arnold. Software Change Impact Analysis. *IEEE Computer Society Press*, 1996.

[4] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: an approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering (TSE)*, 23(7), 1997, pp. 437–444.

[5] M. B. Cohen, M.B. Dwyer, and J. Shi, Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach, *IEEE Transactions on Software Engineering (TSE),* 34(5), 2008, pp. 633-650.

[6] M. B. Cohen, J. Snyder, and G. Rothermel. Testing across configurations: Implications for combinatorial testing. *ACM SIGSOFT Software Engineering Notes*, 2006, pp. 1-9.

[7] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4), 2005, pp. 405–435.

[8] N. Dor, T. Lev-Ami, S. Litvak, M. Sagiv, and D. Weiss. Customization change impact analysis for ERP professionals via program slicing. *International symposium on Software Testing and Analysis (ISSTA)*, 2008, pp. 97-108.

[9] S. Fouche, M. B. Cohen, and A. Porter. Towards incremental adaptive covering arrays. *Symposium on the Foundations of Software Engineering (FSE)*, 2007, pp. 557–560.

[10] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. *IEEE Transactions on Software Engineering (TSE)*, 10(2), 2001, pp. 184–208.

[11] D. Kuhn, D. R. Wallace, and A. M. Gallo. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering (TSE)*, 30(6), 2004, pp. 418–421.

[12] R. O. Kuehl. Design of Experiments: Statistical Principles of Research Design and Analysis. Brooks/Cole, 2000.

[13] M. Nita and D. Notkin. White-box approaches for improved testing and analysis of configurable software systems. *International Conference on Software Engineering (ICSE)*, 2009, pp. 307-310.

[14] A. Nanda, S. Mani, S. Sinha, M.J. Harrold, and A. Orso. Regression testing in the presence of non-code changes. *International Conference on Software Testing (ICST)*, 2011, pp. 21-30.

[15] A. Orso, T. Apiwattanapong, J. Law, G. Rothermel, and M. J. Harrold. An empirical comparison of dynamic impact analysis algorithms. *International Conference on Software Engineering (ICSE)*, 2004, pp. 491-500.

[16] X. Qu, M. B. Cohen, and K. M. Woolf. Combinatorial interaction regression testing: a study of test case generation and prioritization, *IEEE International Conference on Software Maintenance (ICSM)*, 2007, pp. 255-264.

[17] X. Qu, M.B. Cohen, and G. Rothermel. Configuration-aware regression testing: an empirical study of sampling and prioritization, *International Symposium on Software Testing and Analysis (ISSTA),* 2008, pp. 75-85.

[18] B. Robinson and L. White. Testing of user-configurable software systems using Firewalls. *International Symposium on Software Reliability Engineering (ISSRE)*, 2008, pp. 177-186.

[19] E. Reisner, C. Song, K-K. Ma, J. S. Foster, and A. Porter. Using symbolic evaluation to understand behavior in configurable software systems. *International Conference on Software Engineering (ICSE)*, 2010, pp. 445-454.

[20] G. Rothermel, R. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering (TSE)*, 27(10), 2001, pp. 929–948.

[21] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering (TSE)*, 22(8), 1996, pp. 529–551.

[22] G. Rothermel and M. J. Harrold. Empirical studies of a safe regression test selection technique. *IEEE Transactions on Software Engineering (TSE)*, 24(6), 1998, pp. 401–419.

[23] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: A tool for change impact analysis of Java programs. *ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2004, pp. 432-448.

[24] K.D. Scheidemann. Optimizing the Selection of representative configurations in verification of evolving product lines of distributed embedded systems. *Software Product Line Conference (SPLC)*, 2006, pp.75-84.

[25] M. Weiser. Program slicing. *International Conference on Software Engineering (ICSE)*, 1981, pp. 439-449.

[26] C. Yilmaz, M. B. Cohen, and A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Transactions on Software Engineering (TSE)*, 31(1), 2006, pp. 20–34.

[27] I.-C. Yoon, A. Sussman, A. Memon, and A. Porter. Effective and scalable software compatibility testing. *International symposium on Software Testing and Analysis (ISSTA),* 2008, pp. 63–74.

[28] *CodeSurfer*. GrammaTech Inc. http://www.grammatech.com/products/codesurfer

[29] B. Moolenaar. Lavasoft support forums. http://www.lavasoftsupport.com

[30] Free Software Foundation. *gcov*. http://gcc.gnu.org/onlinedocs/gcc/Gcov.html, 2007.

[31] GNU *make*. http://www.gnu.org/software/make/