Configuration Selection Using Code Change Impact Analysis for Regression Testing

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— **Configurable systems that let users customize system behaviors are becoming increasingly prevalent. Testing a configurable system with all possible configurations is very expensive and often impractical. For a single version of a configurable system, sampling approaches exist that select a subset of configurations from the full configuration space for testing. However, when a configurable system changes and evolves, existing approaches for regression testing select all configurations that are used to test the old versions for testing the new version. As demonstrated in our experiments, this *retest-all* approach for regression testing configurable systems turns out to be highly redundant. To address this redundancy, we propose a configuration selection approach for regression testing. Formally, given two versions of a configurable system, *S* (old) and *S′* (new), and given a set of configurations C_S for testing *S*, our approach selects a subset $C_{S'}$ of C_S for regression testing *S′*. Our study results on two open source systems and a large industrial system show that, compared to the retest-all approach, our approach discards 15% to 60% of configurations as redundant. Our approach also saves 20% to 55% of the regression testing time, while retaining the same fault detection capability and code coverage of the retest-all approach.**

Keywords- Configurable System Testing; Configuration Selection; Regression Testing; Static Program Slicing; Change Impact Analysis.

I. INTRODUCTION

Just as software has become essential in our daily activities, so has the ability to configure or customize it. Users *configure* the software by setting their own application preferences ranging from features that are cosmetic to features that modify the system behavior.

It is well known that testing a configurable system under different configurations exposes different faults [11][18][25]. Hence, it is desirable to test a configurable system exhaustively with all possible configurations. But exhaustive testing of configurations is usually infeasible [10]. Recent work has shown that approaches such as *Combinatorial Interaction Testing* (CIT) can improve the cost-effectiveness of testing a single version of a configurable system by *sampling* the full configuration space [11][15][18][25]. Let *S* be a version of a configurable system with a test suite *T* and let C_S be a sampled set of configurations for testing *S*, which is generated by the CIT approach. Our previous work [17] shows that running the full test suite *T* under each configuration in C_S for testing *S* is highly redundant. To address this redundancy, our previous work introduced a *test case selection* approach – given an existing test suite *T* used for testing *S* under a configuration $C \in C_S$, our previous approach [17] selects a subset $T' \subseteq T$ for testing *S* under a different configuration $C' \in C_S$.

However, the challenges of testing a configurable system also persists in later stages of the software lifecycle, where new release of the system (denoted as *S'*) must be *regression tested.* Let $C_{S'}$ be the set of configurations that will be used for regression testing *S'*. Existing configuration selection approaches for regression testing select all configurations that are used to test the old versions for testing the new version, i.e., $C_{S'} = C_S$. Such *retest-all* configuration selection approach for regression testing, however, can be expensive, in spite of using the CIT sampling approach to select a subset of configurations from the full configuration space. For example, as demonstrated in our previous work [18], when the CIT sample C_S was used for regression testing a target system, each configuration required eight hours to regression test, and regression testing of all 60 configurations in the CIT sample C_S required almost three weeks.

To address this problem, our previous work [18] introduced a *configuration prioritization* approach to reorder the configurations to be regression tested. The prioritization approach improves the *rate* of fault detection, but it neither discards redundant configurations nor detects all the faults detected by the retest-all approach. In this paper, we introduce a *configuration selection* approach that selects a small set of configurations from C_S while still retaining the *fault detection capability* (measured by the number of detected faults) of the retest-all approach. Formally, given two versions of a configurable system, *S* (old) and *S′* (new), and given a set of configurations C_S for testing *S*, our approach selects a subset $C_{S'} \subseteq C_S$ of configurations for regression testing *S′*. Our configuration selection approach uses slicing-based code change impact analysis [4] to assist configuration selection. As demonstrated in our study (Section IV.C), our approach can largely reduce the redundancy of the retest-all approach without sacrificing the fault detection capability.

The state of the art in configurable system testing is summarized in TABLE I. "NA" in the table indicates that the prioritization approaches are not applicable for single version systems. As shown in this table, regression testing has been extensively researched at the test case level [21][22][23], whereas the problem of regression testing at the configuration level has received very less attention. Our previous work [20] and the approach proposed in this paper (highlighted in the table) addresses this lack.

TABLE I. THE STATE OF THE ART IN CONFIGURABLE SYSTEM TESTING

Problems		Single Version Testing	Regression Testing
Configuration Level	Selection	[11][15][18][25]	Focus of this paper
	Prioritization	NA	[18]
Test Case Level	Selection	[17]	[22][23][a]
	Prioritization	NA	[21] [a]

a. Regression testing techniques at the test case level are applied on a per-configuration basis for configurable software system.

Overall, our paper makes the following contributions:

- We introduce the first configuration selection approach for regression testing configurable systems.
- We evaluate our approach on two open source systems and a large industrial software system. Our study results show that, compared to the *retest-all* approach, our approach discards 15% to 60% configurations as redundant, and can save 20% to 55% of the testing time, while retaining the same fault detection capability and code coverage of the retest-all approach.

The rest of the paper is organized as follows. Section II presents the background required for understanding our approach. Section III describes our configuration selection approach using an illustrative example. The study design and the results are presented in Section IV. Finally, Section V discusses the related literature and Section VI concludes our paper with pointers to future work.

II. BACKGROUND AND NOTATIONS

In this section, we provide the basic background on configurable systems (Section II.A) and configurable system testing (Section II.B). Particularly, *configuration generation* using the Combinatorial Interaction Testing (CIT) sampling approach [9][11][15] is described (Section II.B.1)). We also introduce *static program slicing* (Section II.C), which is used by our approach for analyzing the impact of code changes. The notations introduced in this section will be used throughout the rest of the paper.

A. Configurable Systems and Configurations

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, *Internet Explorer (IE)*, a popular web browser, is a configurable system. In *IE*, users can select different values (or settings) for the configurable option, *privacy*, from *allow all cookies* to *block all cookies*.

Given a configurable system S (we use S to denote both the configurable system S and its source code), 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*, denoted as C. In this paper, we use the term configuration and configuration instance interchangeably. For each P_i, let $|P_i|$ be the number of values that the users can choose for this option. Accordingly, each possible value of option P_i is denoted as p_{ij}, where $j \in [1, |P_i|]$. For each configurable option P_i, let p_i^C, chosen from $\{p_{ij}, j \in [1, |P_i|]\}$, represent the value assigned to P_i in configuration C. Hence, the configuration instance C can be represented by the set $\{p_1^C, p_2^C, \dots p_m^C\}$. A collection of $|C_S|=k$ such configuration instances, $\{C_1, C_2, C_3 \dots C_k\}$, used for testing S, is denoted as C_S.

B. Configurable System Testing

Effectively testing a configurable system requires that the system is tested with its test suite under different configurations [11][18][25]. Hence, approaches for testing configurable systems consider both configurations and test cases. With hundreds or even thousands of configurable options in practice, it is not possible to exhaustively test a system with all possible configuration instances. To address this problem, testing approaches use *configuration generation* to sample a subset of configuration instances from the set of all possible configurations for testing.

Combinatorial Interaction Testing (CIT) [9] is a systematic and an automated approach for the configuration generation process. CIT has been shown to be superior to both random and exhaustive configuration generation approaches [18][19]. Empirical studies [11][15] on real configurable systems have shown that CIT is effective in generating configurations for testing, measured by the fault detection capability. Yilmaz et al. [25] have shown that configurations generated by the CIT approach can detect configuration-dependent failures and characterize faults more efficiently, compared to the exhaustive generation approaches. CIT is also a widely used technique in the industry [20].

1) The CIT Approach

For a given configurable system, the CIT approach models its configurable options and their associated values, and combines them systematically so that all combinations of values for each t-way ($t > 1$) combination of options are tested together [9]. Here, t is called the strength of testing, and the case when t=2 is called *pair-wise* testing.

TABLE II shows a partial set of configurable options and their values for the text editor *vim*, a configurable system. There are four configurable options $P=\{P_1, P_2, P_3, P_4\}$. P_1, P_2, and P_3 are binary options (with two possible values), i.e., $|P_1|=|P_2|=|P_3|=2$. P_4 has three possible values (i.e., $|P_4|=3$), with $p_{41}=0$, $p_{42}=2$, and $p_{43}=78$. A total of $|P_1|\times|P_2|\times|P_3|\times|P_4|= 2\times2\times2\times3= 24$ configuration instances are possible. Exhaustively testing all possible configuration instances is infeasible for systems in practice, a problem that is addressed by the CIT approach.

TABLE III shows the set $C_S=\{C_1, C_2, \dots, C_6\}$ of configurations generated by the pair-wise CIT approach, which requires that all possible value combinations of each pair of options (totally ${}^4C_2$=6 pairs of options) should appear in at least one of the configurations.

For example, there are $|P_1|\times|P_2|$= 4 value combinations for the pair (P_1=*background*, P_2=*autoread*), which are (*dark*, *ar*), (*dark*, *noar*), (*light*, *ar*), and (*light*, *noar*). In TABLE III, the shaded boxes denote these four value combinations. Each of these value combinations should be covered by at least one configuration. The value pair of (*dark*, *ar*) is covered in C_1, (*light*, *noar*) is covered in C_2, (*dark*, *noar*) is covered in C_3, and (*light*, *ar*) is covered in C_4. It is easy to verify from TABLE III that for the other 5 pairs, (*background*, *tabstop*), (*background*, *textwidth*), (*autoread*, *tabstop*), (*autoread*, *textwidth*), and (*tabstop*, *textwidth*), all value combinations are covered in at least one of the six configurations. But as shown in our previous work [18], even if the number of configurations generated by the pair-wise CIT approach is much smaller than the number of all possible configurations, it is still very expensive (and redundant, as our experiments demonstrate in Section IV.C) to rerun all configurations generated by the CIT approach, especially in a regression testing environment with time and resource constraints.

TABLE II. CONFIGURABLE OPTIONS AND VALUES OF *VIM*

Options	***P_1***	***P_2***	***P_3***	***P_4***
	background	**autoread**	**tabstop**	**textwidth**
Values of Options	dark	ar	8	0
	light	noar	1	2
	▨	▨	▨	78

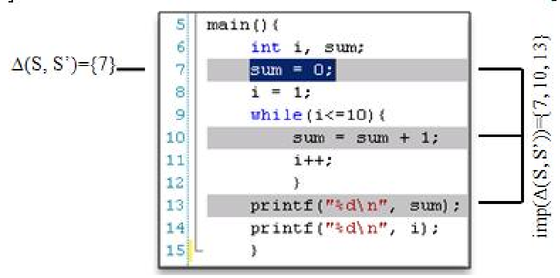
TABLE III. CONFIGURATIONS GENERATED BY PAIR-WISE CIT FOR *VIM*

	P_1	***P_2***	***P_3***	***P_4***
	background	**autoread**	**tabstop**	**textwidth**
C_1	dark	ar	8	78
C_2	light	noar	1	78
C_3	dark	noar	1	2
C_4	light	ar	1	0
C_5	light	ar	8	2
C_6	dark	noar	8	0

C. Static Program Slicing

Static program slicing, first introduced by Weiser [24], refers to the computation of *program points* that effect or are affected by a given program point. The *forward slice* of a program point includes all the program points in the forward control flow affected by the computation or conditional test at the program point. Program points are the most basic fragments of the source code. A program may contain multiple files, a file may contain multiple functions, a function may contain multiple lines, and a line may contain multiple program points. A change or a change block can be considered as a set of program points.

Let *S* be a configurable system and suppose that the configurable system evolves from *S* to a new version, *S'*. The source code of *S'* is shown in Figure 1. Let us assume that only Line 7 has changed between versions *S* and *S'*. Let $\Delta(S, S')$ denote the changes in the code between *S* and *S'*. In Figure 1, $\Delta(S, S')=\{7\}$. Let $imp(\Delta(S, S'))$ represent the code that is statically impacted by the change, $\Delta(S, S')$. In *S'*, the forward slice of Line 7 includes lines 7, 10, and 13 (highlighted in the figure), i.e., $imp(\Delta(S, S'))=\{7, 10, 13\}$. Since the impact of the changed code trivially includes the changed code itself, $\Delta(S, S') \subseteq imp(\Delta(S, S'))$. For our example code in Figure 1, we measure the change and its impact at the line granularity. In this paper, however, we measure the change and its impact at the function granularity.



```
5  main(){
6      int i, sum;
7      sum = 0;
8      i = 1;
9      while(i<=10){
10         sum = sum + 1;
11         i++;
12     }
13     printf("%d\n", sum);
14     printf("%d\n", i);
15 }
```

Figure 1. The forward slice of *sum=0*.

III. OUR APPROACH

In this section, we describe our approach of selecting configurations for regression testing configurable systems. In Section III.A, we describe the problem of configuration selection using a simple example, which will also be used to illustrate our approach. In Section III.B, we describe the various steps in our approach. In Section III.C, we provide the implementation details of our approach.

A. Example and Problem Description

Suppose *S* is a simple configurable system as shown in Figure 2. *S* contains eight functions f_1 to f_8. *S* has three configurable options, i.e., $P=\{P_1, P_2, P_3\}$. As shown in TABLE IV, each configurable option is binary and may be either *True* or *False*, i.e., $|P_1|=|P_2|=|P_3|=2$, $p_{11}=p_{21}=p_{31}=True$ and $p_{12}= p_{22}= p_{32}=False$. A given configurable option of system *S*, P_i, can be mapped to its corresponding *configuration variables* (usually used to receive the values of the options from the user) in the source code of *S*. In our example, each of the configurable options P_1, P_2, and P_3 are mapped to one configuration variable of the same name (unsigned global integers P_1, P_2, and P_3 in lines 26 to 28) in the source code. These configuration variables, and hence their corresponding configuration options, control the different executions of *S*. Let $imp(P_i)$ represent the code in *S* that is statically impacted by the configuration variables corresponding to P_i. As we measure the impact at the function granularity, $imp(P_i)$ is a set of function calls along the static paths in *S* that are impacted by P_i.

The Control Flow Graph (CFG) of *S* is shown in Figure 3. In the CFG, a dashed arrow indicates a function call. For example, functions f_2 and f_6 call the function f_1. The configurable options (variables), highlighted by shaded diamonds in the CFG, determine which static paths are taken in the source code. Depending on the values supplied for the three configurable options, seven static paths (e_1 to e_7) are possible in our example code. In our example, static paths e_1, e_2, e_5, e_6 and e_7 are impacted by the configurable option (variable) P_1, e_6 and e_7 by P_2, and e_3 and e_4 by P_3.

```
int f1(int x){
     return ++x;
}

int f2(int x){
     int s = -f1(x);
     return s;
}

int f6(int x){
     int s = f1(x)%4;
     return s;
}

void f3(){ printf("f3"); }

void f4(){ printf("f4"); }

void f5(){ printf("f5"); }

void f7(){ printf("f7"); }

void f8(){ printf("f8"); }

//configurable options
unsigned int P1;
unsigned int P2;
unsigned int P3;
```

```
void main(){
  int x;
  if (x == 0) {
    if(P1)
      f1(1);
    else
      f2(2);
   } // end x==0

   else {  // x != 0
     f3();
     if(x < 0){
        f5();
        if(P3)
           f4();
     }  // end x < 0

     else{   //x > 0
        if(P1){
           if(P2)
              f8();
           else
              f7();
        }
        else
           f6(6);
     }  // end x > 0

   }  // end x != 0
}
```

Figure 2. The source code of our example configurable system *S*.

TABLE IV. CONFIGURABLE OPTIONS AND VALUES OF *S*

Options	P_1	P_2	P_3
Values of Options	*True*	*True*	*True*
	False	*False*	*False*

For *S*, a total of $|P_1|\times|P_2|\times|P_3|=8$ configuration instances are possible. We applied pair-wise CIT [11][15] to generate a subset of four configurations for testing the initial version of the system *S*. The configuration set generated by pair-wise CIT, $C_S=\{C_1, C_2, C_3, C_4\}$, is shown in TABLE V. Though CIT is used in this example and our experiments in this paper, our selection approach does not require that the original set of configurations be generated by CIT.

Let us assume that a new version of our example system, *S'*, has been released. Let us also assume that only the source code of the function f_1 has been modified between *S* and *S'*, i.e., $\Delta(S, S')=\{f_1\}$. The problem we seek to solve in this paper is to select a set of configurations $C_{S'}$ from $C_S=\{C_1, C_2, C_3, C_4\}$ for testing *S'*. A naïve approach is the *retest-all* approach, which is to regression test *S'* with all four configurations, i.e., $C_{S'} = C_S$. The primary advantage of the retest-all approach is that $C_{S'}$ achieves the best possible code coverage and fault detection capability.

However, there are several disadvantages with the retest-all approach. Though the retest-all approach may scale for small systems such as the one in our illustrative example, the retest-all approach is impractical for large industrial systems with hundreds or even thousands of configurations. As shown in our previous work [18], it is still very expensive to retest all the configurations, even if it is a relatively small set, generated by the pair-wise CIT approach, especially in a regression testing environment with time and resource constraints. Furthermore, some configurations involve heavy setup overhead, which will add substantial additional testing cost. Finally, as demonstrated by our experimental results presented in Section IV.C, the retest-all approach turns out to be highly redundant – in most cases it is possible to achieve the same fault detection capability of the retest-all approach with a small subset of C_S. Therefore, given two versions of a configurable system, *S* and *S'*, and given a set of configurations C_S to test the old version *S*, the problem addressed by this paper is to select a small subset $C_{S'}$ of configurations in C_S that retains the code coverage and regression fault detection capability of the retest-all approach.

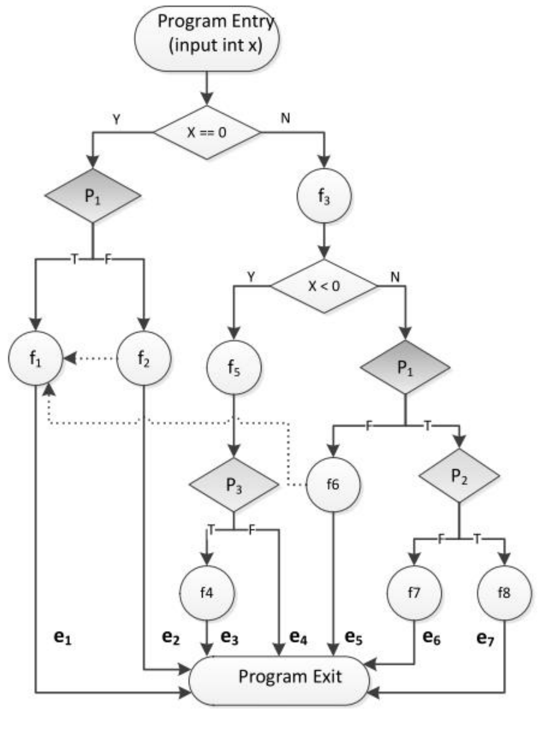


Figure 3. The CFG of our example configurable system *S*.

TABLE V. C_S GENERATED BY PAIR-WISE CIT FOR TESTING *S*

	P_1	P_2	P_3
C_1	*True*	*True*	*True*
C_2	*True*	*False*	*False*
C_3	*False*	*True*	*False*
C_4	*False*	*False*	*True*

Random-selection is another naïve approach to select a subset of C_S. However, as we demonstrate in Section IV.C, the subset of configurations selected by the random-selection approach may not retain the regression fault detection capability of the retest-all approach. In the next section, we introduce our slicing-based configuration selection approach.

B. Description of Our Approach

In this paper, we assume that *P'*, the set of configurable options for *S'*, is same as *P*, the set of configurable options for *S*, i.e., *P'*=*P*. Configuration-aware regression testing in cases where *P'* is a superset or subset of *P* involves configuration *augmentation* or *reduction* besides selection. Configuration augmentation and reduction techniques are outside the scope of this paper and we plan to explore them as a part of our future work.

The goal of regression testing is to retest all the changes in the code (i.e., $\Delta(S,S')$) and the code impacted by these changes (i.e., $imp(\Delta(S,S'))$), to ensure that the code changes do not introduce new faults (regression faults). Let *R* denote the code to retest, i.e., $R=\Delta(S,S') \cup imp(\Delta(S,S')=imp(\Delta(S,S')$, since $\Delta(S,S') \subseteq imp(\Delta(S,S'))$. Consequently, the goal of our configuration selection approach for regression testing is to select only those configurable options (and hence, configurations) that impact the static paths in *R*.

The high-level overview of our approach is shown in Figure 4. Our approach has two main steps, *selecting configurable options* and *selecting configuration instances*, as shown by the dotted boxes in the figure. First, among all the configurable options in *P'* of *S'*, our approach selects a subset of configurable options $P'_{sel} \subseteq P'$, that impacts the static paths in *R*. Only the configurable options in P'_{sel} need to be retested for *S'*. Second, our approach selects a subset $C_{S'}$ of configurations from C_S that will cover the pair-wise interactions between the selected configurable options in P'_{sel}. We use pair-wise interaction as the criteria for CIT because it is the most prevalent CIT criteria for configuration generation. The final set of configurations selected by our approach, $C_{S'}$, is used for regression testing the new version *S'*. In following sections, we illustrate each step in our approach using the example introduced in the previous section.

1) Selecting Configurable Options

First, our approach identifies the impact of the changed code, i.e., $R=imp(\Delta(S,S')$. Next, our approach selects a set of configurable options that impacts the static paths in *R*. A configurable option is selected if its impact intersects with the impact of the changed code, *R*. Formally, a configurable option $P_i \in P'$ is selected by our approach if

$$imp(P_i) \cap R \neq \emptyset \text{ --- (1)}$$

In this paper, we measure the impact at the function granularity. Recall that the impact of a configurable option of *S* is the set of function calls along the static paths in *S* that are impacted by the configurable option. For example, as shown in Figures 2 and 3, P_1 impacts the static paths e_1, e_2, e_5, e_6 and e_7. Since functions f_1, f_2, f_6, f_7 and f_8 are called along these paths, we regard these functions as the functions impacted by P_1. In our illustrative example, the impacted functions of each configurable option are: $imp(P_1) = \{f_1, f_2, f_6, f_7, f_8\}$, $imp(P_2) = \{f_7, f_8\}$, and $imp(P_3) = \{f_4\}$.

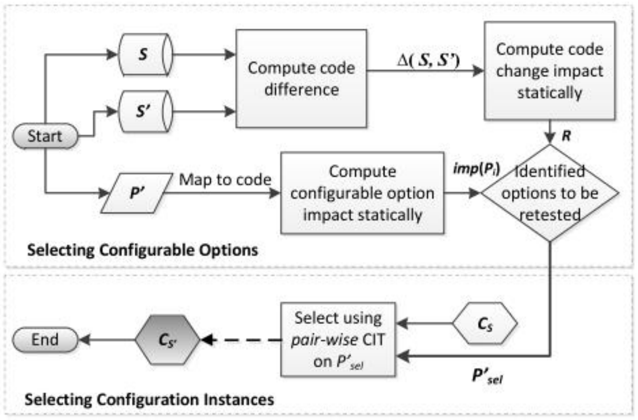

S
S'
Start
Compute code difference
Δ(S, S')
Compute code change impact statically
R
P'
Map to code
Compute configurable option impact statically
imp(P_i)
Identified options to be retested
Selecting Configurable Options
End
C_S'
Select using pair-wise CIT on P'_sel
C_S
P'_sel
Selecting Configuration Instances

Figure 4. The high-level overview of our approach.

As introduced in Section A, in our example, we have assumed that $\Delta(S, S')=\{f_1\}$. It is easy to manually verify that the functions f_1, f_2 and f_6 are statically impacted by the code changes in f_1, i.e., $R=imp(\Delta(S, S'))=\{f_1, f_2, f_6\}$, which is the impact of the code that has changed between *S* and *S'*. Based on (1), our approach selects only P_1 (highlighted in the first column of TABLE IV) for regression testing (i.e., $P'_{sel} = \{P_1\}$), because only the impact of P_1 intersects with the impact of the code changes. As a result, only functions f_1, f_2 and f_6, called along static paths e_1, e_2 and e_5 are to be retested in *S'*, satisfying the goal of regression testing.

2) Selecting Configuration Instances

In this section, we describe how our approach selects configuration instances from C_S to construct $C_{S'}$ after selecting the configurable options P'_{sel} in the first step.

Pair-wise CIT is the most prevalent technique to select configurations from the full configuration space. As introduced in Section II.B.1, the pair-wise criterion requires that all value combinations of each pair of configurable options should be covered at least once in the selected configurations to be tested. For our regression selection purpose, only the interactions between the selected options in P'_{sel} are important. Accordingly, all value combinations of each pair of *selected* configurable options (P'_{sel}) should be covered at least once in the selected configurations to be retested. All other options not in P'_{sel} (i.e., $P' - P'_{sel}$) may use any of its values randomly.

Our algorithm for selecting configuration instances is shown in Figure 5. First, as shown in Line 1, the algorithm constructs the *value pair* set *A* that consists of all the value combinations of all pairs of options selected in P'_{sel}. Suppose $P'_{sel} = \{P_1, P_2\}$, there will be four value pairs as $A = \{(p_{11}, p_{21}), (p_{11}, p_{22}), (p_{12}, p_{21}), (p_{12}, p_{22})\}$. In our illustrative example, P'_{sel} only contains one option, P_1, and hence, only two value pairs should be covered in the selected configurations, i.e., $A=\{(p_{11}), (p_{12})\}$ where p_{11}= *True* and p_{12}=*False*. In other words, there must exist at least one configuration where the value of P_1 is *True* and at least one configuration where the value of P_1 is *False*.

Next, the algorithm starts selecting configurations until one of the two stopping criteria is met: (1) all value pairs in A are covered, i.e., the set of remaining pairs, Z, is empty (the while loop) OR (2) no remaining configurations in C_S can cover any of the remaining value pairs in Z (lines 7 and 8). The second stopping criteria is satisfied if the original set of configurations C_S is not created by the CIT approach, which indicates that some important value pairs are missing in C_S. In this case, we can augment $C_{S'}$ using the regular CIT approach, which will be addressed in our future work.

Another key point of this algorithm is, in each turn of selection (the while loop), the algorithm always selects a configuration that would cover the most value pairs that have not been covered yet, i.e., value pairs that are still in Z: (1) In Line 5, our algorithm counts the number of *uncovered* value pairs in each $C_i \in C_S$, denoted as N_i (i = 1 to | C_S |); (2) In Line 6, our algorithm finds the largest N_i; (3) In Line 9, our algorithm selects the configuration whose N value is the largest. When there is a tie for selection among configurations, the algorithm randomly selects one of them. For the configurable options that are not selected in the first step (Section B.1), instead of picking a random value, our approach reuses the same value used for testing the previous version of the program (as instantiated in selected configurations). Moreover, the chance of a tie and the number of configurations for random selection in the event of a tie decreases with the number of iterations of the while loop.

In our example, during the first loop, N_1=N_2=N_3=N_4=1. Suppose C_2 is randomly selected from the four candidate configurations, Z is updated to Z={(p_{12})}. During the second loop, N_1=0 and N_3=N_4=1. Suppose C_3 is selected from the two candidate configurations, Z becomes Ø. As a result, our approach selects $C_{S'}$ = {C_2, C_3}, which saves us 50% of the regression testing time, compared to the retest-all approach.

```
1.  set $A$ = all the value combinations of all ${}^{|P'_{sel}|}C_2$ pairs of
    options in $P'_{sel}$
2.  set $Z = A$
3.  $C_{S'} = Ø$
4.  while $Z \neq Ø$ do:
5.      $N_i$ = the number of value pairs that are covered by
        configurations $C_i \in C_S$ and are also in $Z$ ($i$ = 1 to $|C_S|$)
6.      $N_{max}$ = the largest value of $N_i$
7.      if $N_{max} = 0$
8.          break
9.      Select a configuration $C_k \in C_S$ if $N_k = N_{max}$
10.     $C_{S'} = C_{S'} \cup C_k$
11.     $C_S = C_S - C_k$
12.     $Z = Z$ – (value pairs in $C_k$)
```

Figure 5. Algorithm for selecting configuration instances in our approach.

C. Implementation of Our Approach

Our approach employs a static analysis tool called *CodeSurfer* [27] for computing the configurable option impact and the code change impact. *CodeSurfer* uses forward program slicing to compute $imp(P_i)$ and $imp(\Delta(S,S'))$. 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. In our experiments, we conservatively use the most expensive *CodeSurfer* option available for them (for example, for pointer analysis, we use the Andersen's algorithm [2]). The most expensive parameter setting of *CodeSurfer* is denoted as H (shown in TABLE VI).

As shown in our study (Section IV.B), there is one static analysis parameter, *non-local analysis*, which may impact the preciseness of our approach. 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 is configurable and can be shut off. The H setting of *CodeSurfer* with non-local analysis turned off is denoted as the setting L.

TABLE VI. THE *H* PARAMETER SETTINGS FOR *CODESURFER*

Static Analysis Parameter	Settings
Non-local analysis	Yes
Dependences	Inter/intra Control and Data
Pointer analysis	Andersen's algorithm [2]
Summary edges	Yes

The other components of our approach, including the algorithm shown in Figure 5, are implemented as C/C++ programs.

IV. EVALUATION OF OUR APPROACH

In this section, we present our results of evaluating our approach based on three criteria. First, we study the *quality* of the configurations selected by our approach for regression testing in terms of the fault detection capability (measured by the number of faults detected) and the code coverage. Second, we study what percentage of configurations is discarded as redundant by our approach for regression testing. Finally, we study the overall regression testing *time-savings* possible with our approach. The research questions for our evaluation are as follows:

- **RQ1**: Can the set of configurations selected by our approach achieve the same fault detection capability and code coverage as the retest-all approach? How does it work compared to the random-selection?
- **RQ2**: What percentage of configurations is discarded as redundant by our approach for regression testing?
- **RQ3**: How much regression testing time can our approach save?

In Section IV.A, we describe the subjects that are used for our evaluation. In Sections IV.B-IV.D, we provide the details from a series of experiments we conducted to evaluate our research questions.

A. Study Subjects

We first evaluate our approach on *grep* and *make*, two open source packages written in C. *grep* is a command-line utility for searching plain-text data sets for lines matching a regular expression. *make* is a widely popular and general purpose tool used to compile programs. The source code of *grep* and *make* are available at the GNU website [29][30]. We analyzed two versions of *grep*, *grep-1.0* as the original and old version and *grep-2.0* as the changed and new version. *grep* contains about 8,000 uncommented lines of code. We analyzed two versions of *make*, *make-3.77* as the original and old version and *make-3.78.1* as the changed and new version. *make* contains about 15,000 uncommented lines of code. We demonstrate our approach for selecting a set of configurations for regression testing the new versions of *grep* and *make*.

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 about 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.

B. Study 1: Fault Detection Capability and Code Coverage

1) Study Design

make-3.78.1 is a subsequent version of *make-3.77*, released 14 months after *make-3.77*. 26 out of 28 source files changed from *make-3.77* to *make-3.78.1*. Spread across the 26 changed files, there are 869 changed blocks of code. The code change blocks were computed by *WinMerge* [31], an open source code differencing and merging tool for Windows. We assume that a system testing involving configurations was conducted every month. We also assume that the 869 changes made through the 14 months were evenly distributed in each month. Hence, we randomly selected 60 ($869/14 \approx 60$) changes to form an intermediate version of *make*, *make-3.77.b*, between *make-3.77* and *make-3.78.1*. By assuming that 25% of the changes may introduce regression faults, we randomly seeded 15 faults in the 60 selected changed blocks of *make-3.78.1*. The seeded faults include the hand-seeded faults obtained from the Software Infrastructure Repository (SIR) [12] and the mutations generated by the *mutgen* tool [3].

grep-2.0 contains eight source files. Since there is no version before *grep-2.0* that is publicly available (we had problems building the later versions of *grep* with *CodeSurfer*), we randomly selected 15 blocks in *grep-2.0* as changes from a "virtual" previous version *grep-1.0*. We used *mutgen* to generate more than 10000 mutants spread across all the files. We then randomly selected a compilable mutant for each selected changed block. Hence, in total, there were 15 faults seeded in *grep-2.0*.

We selected 11 configurable options for *make* and 14 options for *grep*. All these options are binary options. Then, for each subject, we created the initial set of configurations (i.e., C_S) from these options using the CIT approach [10][11]. The numbers of configuration instances in C_S for testing the old version were 14 and 7 for *make* and *grep* respectively. The test suites for *make* and *grep* were both obtained from the SIR [12].

2) Indepednet and Dependent Variables

The independent variable for RQ1 is the set of configurations (i.e., $C_{S'}$) selected for regression testing *make-3.78.1* and *grep-2.0*. We compare our approach to the retest-all and the random-selection approaches. Hence, the values of the independent variable are:

- The set of configurations selected by the retest-all approach, which is the full set of configurations from the original version (C_S)
- The set of configurations selected by our approach
- The set of configurations selected by the random-selection approach

As introduced before, an important factor in examining the quality of a regression selection approach is the *fault detection capability* (FD, measured by the number of faults detected) of the selected configurations. We select it as our first dependent variable. Besides this, we also examine the code coverage capability of the selected configurations, particularly the coverage of the code to retest (i.e. $R=imp(\Delta(S,S'))$). We call it *changed function coverage* (CFC), our second dependent variable. We used *gcov* [28] to collect the code coverage data.

3) Results

First, we discuss the results of applying our approach on *grep*. When we used the most expensive setting *H* of *CodeSurfer* (described in Section III.C), the impact of all 14 configurable options intersects with the impact of changes between *grep-1.0* and *grep-2.0*. This indicates that we have to select all configurable options, which leads to selecting all configurations from C_S. We looked into the actual impact of each configurable option. We found that the impact of each option is the same, which almost covered all the functions in the system. This problem is called *impact explosion* [1][7], mainly caused due to large central global data structures. In *grep,* all options are defined as global variables and organized in a central structure.

Because of this problem, impact computed with the *H* setting of *CodeSurfer* is very imprecise, and hence, not suitable for selecting the configurable options. Therefore, we use a less expensive setting *L* of CodeSufer by turning off the non-local analysis and examine if it will affect the precision of our approach.

The results from analyzing the *grep* package show that, with the *L* setting, among the 14 configurable options, the impact of 4 options does not intersect with any impact of code changes. Hence, 10 out of 14 options are selected for regression testing *grep-2.0*. By applying our configuration selection algorithm (Figure 5) on these selected options, six out of seven configurations are selected, as shown in the shaded box of TABLE VII (the fourth column, containing the shaded box, shows the number of configurations selected by different approaches, i.e., the size of $C_{S'}$).

The fault detection capability (FD) of the sets selected by three different approaches for *grep*, are shown in the second column of TABLE VII. 6 out of 15 faults were detected by the full set of all 7 configurations. The set of configurations selected by our approach detects the same number of faults as the retest-all approach. In contrast, the set of configurations selected by the random-selection approach misses one fault.

By investigating the faults closely, we found that among the six faults that are detected by the full set of configurations, four faults are configuration dependent. Particularly, in the four faults, two faults are related to certain single options and two faults are related to certain pair-wise option combinations/interactions. All these important (fault-revealing) options are selected by our approach. As a result, the fault detection capability of the configurations selected by our approach can be considered to be the same as that of the retest-all approach.

From our results of changed function coverage (CFC), shown in the third column of TABLE VII, it appears that the changed function coverage is same for all three approaches – 52 functions are covered out of 72 changed functions. However, the coverage differences between the three approaches are revealed at the statement granularity, which explains the different fault detection capabilities.

TABLE VII. RESULTS FROM GREP

	FD	CFC	Size of $C_{S'}$	Reduction Rate
Retest-all	6/15	52/72	7	-
Random- selection	5/15	52/72	6	-
Our approach	6/15	52/72	6	**1/7 = 14%**

Next, we discuss the results of applying our approach on *make*. Among the 11 configurable options, only 5 out of 11 options are selected for regression testing *make-3.78.1*. By applying the selection algorithm, 6 out of 14 configurations are selected, as shown in the shaded box in TABLE VIII.

The FD and CFC results of the sets selected by three different approaches for *make* are shown in the second and third columns of TABLE VIII. 8 out of 15 faults were detected by the 14 configurations selected by the retest-all approach and 109 out of 192 changed functions were covered by these selected configurations. The set of configurations selected by our approach achieves the same fault detection capability and changed code coverage as the retest-all approach. In contrast, the set of configurations selected by the random-selection approach misses 5 faults and 26 functions.

TABLE VIII. RESULTS FROM MAKE

	FD	CFC	Size of $C_{S'}$	Reduction Rate
Retest-all	8/15	109/192	14	-
Random-selection	3/15	83/192	6	-
Our approach	8/15	109/192	6	**8/14 = 57%**

In conclusion, the set of configurations selected by our approach can detect the same number of regression faults that are detected by the retest-all approach, and hence, outperforms the random-selection approach.

C. Study 2: Effectivness

1) Study Design

In our study on *ABB1*, we analyzed a development version (called v0) as the original version and a release version (called v1) as the changed version. 70 source files and 203 blocks had changed between versions v0 and v1. Among the 203 changes, we randomly selected three sets of 30 changes for analysis.

There are totally 545 configurable options with *ABB1*. The possible values of these options range from 2 to 9. About 90% of the options are binary options. We used pair-wise CIT [10][11] to generate the initial set of configurations for v0, with one hour time limit for the generation. There were 159 configurations generated.

ABB1 defines the configurable options differently from *make* and *grep*. Instead of defining global variables for configurable options, *ABB1* stores the values of its configurable options in a database. Each time the option is used, a function retrieves the value from the database. Hence, unlike *grep* and *make*, there is no impact explosion problem [1] for *ABB1* and we could use the highest setting *H* (described in Section III.C) of *CodeSurfer* for computing the impact analysis without any loss of precision.

2) Results

First, we discuss the effectiveness of our approach on *grep* and *make*. As shown in the fourth column of Tables VII and VIII, our approach selects six configurations (discards one) for testing *grep-2.0* and selects six (discards eight) for testing *make-3.78.1*. As a result, our approach discards 14% and 57% (shown in bold in the last column) configurations for regression testing them respectively.

TABLE IX. NUMBER OF CONFIGURABLE OPTIONS SELECTED FOR V1

	Change Set 1	Change Set 2	Change Set 3	Average
Retest-all	545			
Selected	167	161	161	163
Reduction	69%	70%	70%	**70%**

TABLE X. NUMBER OF CONFIGURATIONS SELECTED FOR REGRESSION TESTING V1

	Change Set 1	Change Set 2	Change Set 3	Average
Retest-all	159			
Selected	120	120	120	120
Reduction	25%	25%	25%	**25%**

Next, we discuss the results of *ABB1*. TABLE IX shows the number of configurable options that are selected for retesting v1. Among all 545 configurable options, 167 options were selected for regression testing given the first code change set. 161 options were selected given the second and third sets of code changes. The average reduction in configurations is about 70% (shaded box). By using pair-wise CIT with the one-hour time limit, 120 configurations were selected from the full set of configurations, regardless of which set of code change is addressed, as shown in TABLE X. As a result, our approach discards about 25% configurations to be retested.

D. Study 3: Regression Testing Time Savings

In this section, we calculate the regression testing time that is saved by our approach compared to the retest-all approach. The results are shown in TABLE XI (*h*, *m*, and *s* represent hours, minutes, and seconds, respectively).

1) Overheads of Our Approach

The time overhead incurred by our approach (shaded in TABLE XI) has three main components: the *build time* for building the configurable system with *CodeSurfer*, the *slicing time* to compute the impact of code changes and configurable options, and the *selection time* for executing the selection algorithm (Figure 5).

CodeSurfer performs several static program analyses on the source code by transparently integrating with the compile and link stages of the software build. *Build time* is the time required for this build stage. *Slicing time* is the time required to compute the impact set as the forward slice, using the information computed during the build stage. The build times, usually in the order of hours or days, are much longer than slice times, which are usually in the order of a few seconds. Hence, in this paper, we ignore the slice times and only consider build time overheads incurred by change impact analysis. Our experiments were run on a 2GHz quad-core Windows Server 2008 machine with 24GB RAM.

2) Testing Time Savings

The first five rows in TABLE XI show the actual testing time with the selected configurations (i.e., $C_{S'}$) by the retest-all approach and our approach. Particularly, *TestTimePerC* represents the time for testing one single configuration, including the time of setting up the configuration. $|C_{S'}|$ denotes the size of $C_{S'}$, i.e., the number of selected configurations. The time taken by different approaches is the product of these two factors, highlighted in the table in **bold**. As a result, the testing time saved (T_{saved}) by our selection approach is 5 minutes (50%), 387 minutes (55%), and 167 hours (21%) for *grep*, *make*, and *ABB1*, respectively, as shown in the last two rows in TABLE XI.

TABLE XI. REGRESSION TESTING TIME SAVINGS WITH OUR APPROACH

		grep	***make***	***ABB1***
Testing Time with $C_{S'}$ selected by different Approaches	*TestTimePerC*	10 m	50 m	5 h
	$\lvert C_{S'}\rvert$ by *Retest-all*	7	14	159
	Time taken by retest-all (T_{all})	**70 m**	**700 m**	**795 h**
	$\lvert C_{S'}\rvert$ by our approach	6	6	120
	Time taken by our approach (T_{select})	**60 m**	**300 m**	**600 h**
Overhead	*Build time*	14 s	3 m	20 h
	Selection time	5 m	10 m	8 h
	Overhead	5.2 m	13 m	28 h
Testing Time Saving	$T_{saved} = (T_{all} - T_{select})$ - ***Overhead***	5 m	387 m	167 h
	T_{saved} / T_{all}	50%	55%	21%

V. Related Work

A. Configurable System Regression Testing

Approaches for regression testing of configurable systems [13][25][26] study the issue of improving the effectiveness of configuration sampling for testing a new version of system. Our previous work [18] studied the impact of configurations across multiple versions of a system as it evolves. Our previous approach prioritizes the full set of configurations for regression testing the new version of the system, in order to improve the early fault detection rate. However, it suffers from some limitations. First, it requires information from prior versions, which is not always available. Second, it requires that the set of configurations to be prioritized is generated by the CIT approach, which is not always the case. Finally, prioritization approaches do not address the redundancy problem.

Recently, Nanda et al. [16] proposed an approach to support regression testing of systems that exhibit frequent changes in non-code parts of the system. Their approach focuses on changes to databases and simple configuration files but does not consider the code changes between two versions of a system.

B. Regression Test 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'* of *S*. Ad-hoc selection approaches [14] are not safe because they may miss some test cases that reveal faults in the modified program. Instead, *safe* test case selection approaches [22] 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 [23] based on analyzing the Control Flow Graphs (CFG) of *S* and *S'*.

However, traditional regression testing approaches do not consider configurations in testing. In configurable system testing, configuration selection is as important as test case selection. Our approach is a configuration selection approach and is complementary to traditional regression test selection approaches.

There has been some work [5][8] on using program slicing for regression test selection. Slicing-based regression testing approaches can be categorized into two groups [6] – approaches that use static slicing and approaches that use dynamic slicing. There techniques use program slicing on both old and new programs to identify affected statements. Again, these approaches are applicable to regression testing traditional systems wherein the focus is on test case selection. In contrast, our approach applies static slicing for regression testing configurable systems wherein the focus is on configuration selection.

VI. Conclusion

In this paper, we introduce the first configuration selection approach for regression testing configurable systems. Our study results show that, compared to the *retest-all* approach, our approach discards 15% to 60% of configurations as redundant. Our approach also saves 20% to 55% of the regression testing time, while retaining the fault detection capability and code coverage of the retest-all approach.

In our approach, we assume that the set of configurable options is the same for old and new versions. But some changes between versions may add or remove options. In such situations, in addition to configuration selection, configuration augmentation or reduction techniques are also required. We will address these problems in future work.

In this paper, we introduce an approach for configuration selection. In our previous research we introduced an approach for configuration prioritization [18]. We expect that a combination of these approaches may further improve the testing effectiveness and efficiency, compared to either approach alone. Finally, in the subjects we studied, the problem of *impact explosion* [1][7] does not show any impact on the effectiveness of our approach. However, we will investigate the effect of impact explosion in depth as a part of our future work.

ACKNOWLEDGMENT

We thank GrammaTech Inc. for providing technical support for *CodeSurfer*, the static analysis tool used in this paper.

REFERENCES

[1] 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.

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

[3] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, 32(8), 2006, pp. 608–624.

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

[5] D. Binkley. Semantics guided regression test cost reduction. *IEEE Transactions on Software Engineering (TSE)*, 23(8), 1997, pp. 498–516.

[6] D. Binkley. The Application of Program Slicing to Regression Testing. *Information and Software Technology Special Issue on Program Slicing*, 1999, pp. 583–594.

[7] D. Binkley, N. Gold, and M. Harman. An empirical study of static program slice size. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(2), 2007, pp. 2:1-2:32.

[8] S. Bates and S. Horwitz. Incremental program testing usingprogram dependence graphs. *ACM SIGPLAN-SIGACT symposium on Principles of programming languages(POPL)*, 1993, pp. 384–396.

[9] 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.

[10] 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.

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

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

[13] S. Fouché, M.B. Cohen, and A. Porter. Incremental covering array failure characterization in large configuration spaces. *International Symposium on Software Testing and Analysis (ISSTA)*, 2009, pp. 177-187.

[14] 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.

[15] 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.

[16] 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.

[17] X. Qu, M. Acharya, and B. Robinson. Impact analysis of configuration changes for test case selection. *International Symposium on Software Reliability Engineering (ISSRE)*, 2011, pp. 140-149.

[18] 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.

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

[21] 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.

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

[23] 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.

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

[25] 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.

[26] 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.

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

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

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

[30] GNU *grep*. http://www.gnu.org/software/grep/

[31] *WinMerge*. http://winmerge.org/