2013 IEEE Sixth International Conference on Software Testing, Verification and Validation

Oracle-Based Regression Test Selection

Tingting Yu*, Xiao Qu†, Mithun Acharya†, Gregg Rothermel*‡

*Department of Computer Science
University of Nebraska-Lincoln
Lincoln, NE, USA
{tyu|grother}@cse.unl.edu

†Industrial Software Systems
ABB Corporate Research
Raleigh, NC, USA
{xiao.qu|mithun.acharya}@us.abb.com

‡Division of Web Science and Technology
KAIST
Daejeon, South Korea
gregg.rothermel@gmail.com

Abstract — **Regression test selection (RTS) techniques attempt to reduce regression testing costs by selecting a subset of a software system's test cases for use in testing changes made to that system. In practice, RTS techniques may select inordinately large sets of test cases, particularly when applied to industrial systems such as those developed at ABB, where code changes may have far-reaching impact. In this paper, we present a new RTS technique that addresses this problem by focusing on specific classes of faults that can be detected by internal oracles – oracles (rules) that enforce constraints on system states during system execution. Our technique uses program chopping to identify code changes that are relevant to internal oracles, and selects test cases that cover these changes. We present the results of an empirical study that show that our technique is more effective and efficient than other RTS techniques, relative to the classes of faults targeted by the internal oracles.**

I. Introduction

Regression testing is a testing process used to validate modified software and detect whether new faults have been introduced into previously tested code. In practice, regression testing can be expensive. To address this problem, researchers have investigated various strategies, including techniques for regression test selection and test case prioritization ([1] provides a survey).

In this work, we are interested in regression test selection (RTS) techniques. RTS techniques select, from an existing test suite for a system, test cases that are relevant to a modified version of that system, and that are less expensive to execute than the complete suite. While empirical studies of RTS techniques (e.g., [2], [3]) have shown that they can be cost-effective, they have also shown that in certain cases these techniques may yield no benefits. In particular, in large industrial systems such as those developed at ABB, code modifications made for new product versions often have far-reaching impact on the rest of the system's code [4]. In such cases, RTS techniques may select inordinately large numbers of test cases. Approaches for reducing the numbers of test cases selected, while retaining the effectiveness (in terms of fault detection) of the selected test cases, would be helpful.

A second factor that can impact the cost-effectiveness of RTS techniques relates to the manner in which test engineers verify the results of testing. Recent work [5], [6], [7] has shown that there are benefits to paying attention to test oracles when testing. Most work on oracles, however, has focused on using them in test case generation (e.g., [8]), rather than on using them as an aid to regression testing. Recent work has considered the ability of oracles to improve the effectiveness of test case prioritization [9], [10], but has not considered applications to regression test selection.

The most typical approach for verifying test results involves checking system outputs following test execution, a process which, in the regression testing context, often involves "differencing" system outputs with those of prior system versions. In such cases, we say that engineers are using *output-based test oracles*. While output-based oracles can be effective, they can fail to detect faults that do not propagate to system outputs. Such faults can then remain "silent", even in highly tested systems that have been in operation for millions of hours, only to finally manifest themselves under new and previously unexpected operating scenarios [11].

One alternative to output-based oracles involves *internal oracles*; oracles that enforce constraints on system states in an attempt to detect specific classes of faults. Such faults may include, for example, those related to memory management, concurrency, or file operations. Such faults are known for their propensity to be important and their ability to escape detection by output-based oracles [12], [13]. Moreover, the sets of faults targeted by internal oracles (and the choice of oracles to employ) can be tailored to specific applications in accordance with organizational needs.

We believe that by utilizing internal oracles in the regression test selection process, we can address both of the drawbacks of RTS techniques described above; that is, we can reduce the tendency of RTS techniques to select too many test cases, and we can actually enhance the ability of RTS techniques to detect faults – at least for those classes of faults that are targeted by the internal oracles. Internal oracles can then serve as useful complements to output-based oracles. To investigate this belief, we have developed an oracle-based RTS technique. Our technique selects test cases associated with system changes, but only those that are relevant to a set of internal oracles that are known to be important for the system under test. Our technique uses program chopping to locate statements that can affect the output of these test oracles; these statements are then used to direct the selection of test cases.

To assess our oracle-based RTS technique, we conducted an empirical study in which we applied it to two open source

978-0-7695-4968-2/13 $26.00 © 2013 IEEE
DOI 10.1109/ICST.2013.34

systems and several components of a large ABB system. We use oracles studied in our prior work [14], and also oracles that are specific to ABB systems and that are mined as rules directly from source code [15]. We compare our technique to the retest-all technique (in which all test cases are re-executed) and to a code-differencing-based RTS technique. Our results show that our technique substantially improves on these other techniques in terms of efficiency, while equaling or outperforming them in terms of fault detection with respect to faults targeted by the internal oracles.

II. Background

This section provides background information on regression test selection, test oracles, and program analysis techniques. Section VI provides further discussion of related work.

A. Regression Test Selection

Given program P, modified version P', and test suite T, engineers use regression testing to test P'. A retest-all regression testing technique reuses all test cases in T, but given long-running test suites can be expensive. Regression test selection (RTS) techniques (e.g., [3], [16], [17]; select a subset of test cases T' from T for regression testing P'.

B. Test Oracles

The most typical approach to verifying test results involves using *output-based oracles*; that is, oracles that check system outputs. However, faults may escape detection if a test suite fails to propagate their effects to program outputs. In such cases, output-based test oracles are inadequate.

In contrast, *internal oracles* monitor and enforce constraints on internal program states seeking evidence that *infections* have occurred – that is, cases in which program states have been altered in violation of the constraints enforced by the given oracles. These oracles then signal the presence of those infections to alert testers to the possible presence of faults. Previous research has argued that internal oracles can increase the probability of fault detection [18], [19]. Our own recent work [14], [20] introduced a framework for testing embedded systems that provides observability through internal oracles that are used generally by engineers (such as data races, deadlocks, and memory safety). The observations gathered by our internal oracles can be used to detect the presence of faults that might not otherwise be detected.

Whereas the foregoing types of *generic* oracles can be used generally when testing large classes of software systems, another approach to verification relies on internal oracles that are *system-specific*. Sun et al. [15] employ a dependence-based graph mining technique [21] to mine system-specific rules from source code. Their approach identifies frequent code patterns as *graph minors* [22] by analyzing a product's system dependence graph [23]. Their approach then automatically transforms the mined rules into *checkers* for a static analyzer called `Klocwork` [24]. `Klocwork` uses these checkers as oracles to statically find rule violations in the code.

```
1. int main(int argc, char *argv[]){
2.   int i, sum, size;
3.   sum = 0;
4.   size = 32;
5.   i = atoi(argv[1]);
6.   if (i > 1)
7.       i++;                          C
8.   else
9.       size = 16;                    Co
10. char *str = malloc(size);
11. while(i > 5){
12.      sum = sum + size;
13.      i--;
14. }
15. if(strlen(argv[2]) <= 16)
16.      strcpy(str, argv[2]);
17. else
18.      printf("error\n");
19. printf("%s", str);
20. free(str);                         P
21. }
```

```
1. int main(int argc, char *argv[]){
2.   int i, sum, size;
3.   sum = 0;
4.   size = 32;
5.   i = atoi(argv[1]);
6.   if (i > 1)
7.       i--;                          C' = {line7, line9}
8.   else
9.       size = 8;                     C'o = {line9}
10. char *str = malloc(size);
11. while(i > 5){
12.      sum = sum + size;
13.      i--;
14. }
15. if(strlen(argv[2]) <= 16)
16.      strcpy(str, argv[2]);
17. else
18.      printf("error\n");
19. printf("%s", str);
20. free(str);                         P'
21. }
```

Fig. 1. Illustrative example

There are, however, several limitations to static checking of system-specific rules. First, state explosion in static checking may cause scalability problems [25]. Second, static checking suffers from false positives due to imprecise local information and infeasible paths. For example, the analysis engine of `Klocwork` cannot distinguish a structure variable from its field. Third, modern systems are increasingly dependent on hardware and written in diverse languages, and it is difficult for static checkers to analyze all of such systems' source code.

In this work, we utilize system-specific rules mined from source code, but instead of statically checking them, we use them as system-specific oracles to guide regression test selection and dynamically examine the testing results.

C. Program Analysis Techniques

Static program slicing [26] involves the computation of program points of interest that affect or are affected by other program points. The forward slice from a given program point of interest (*s*) includes all program points that context-sensitively follow *s* in forward control flow and are control or data dependent (directly or transitively) on the computation or conditional test performed at *s*. Backward slicing, on the other hand, is computed by context-sensitively tracing dependencies along backward control flow from the point of interest. Program points are basic fragments of 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 or vertices. In this paper, for convenience, we consider program points at the granularity of source code statements.

Static program chopping, roughly speaking, is the intersection of forward and backward slicing. Two points of interest, source (*s*) and target (*t*), are chosen, and the chop consists of statements that are dependent on *s*, and on which *t* is dependent. As such, chopping reveals the ways in which one program point can affect another program point.

We use program P' shown in Figure 1 to illustrate how chopping works. Suppose the statement in line 16 is the target *t* and the statement in line 5 is the source *s* for chopping. In this case, the set of statements at lines $\{5, 6, 9, 10, 16\}$ constitute the chop. This is because variable $size$ in line 9 is control dependent on variable i in line 6, which is computed at *s*, and because $size$, in turn, then affects the computation at t.

Generic Oracles

a	strcpy(strt, strs): sizeof(strs) $\leq$ sizeof(strt)
b	sem o: sem_acq(o) $\Rightarrow$ sem_rel(o)

System-specific Oracles

c	ABB_db1_create() $\Rightarrow$ ABB_db1_close()
d	return ABB_objX: ABB_objX == OK

Fig. 2. Example internal oracles

III. Approach

We now present our oracle-based RTS technique, which for brevity we henceforth refer to as *RTSO*. The goal of RTSO is to select test cases that test program changes while also impacting relevant internal test oracles.

A. Internal Oracles

The internal oracles considered in this work include generic internal oracles [14] (oracles that can be used generally on any system) and system-specific internal oracles (oracles that are specific to a particular system). System-specific internal test oracles can be specified by engineers who are familiar with the software system under test, or can be obtained from the system specifications. We use a pre-existing set of system-specific oracles that were automatically mined by Sun et al. [15] from the source code of a system developed at ABB using an existing graph mining approach. Figure 2 displays examples of generic and system-specific internal oracles.

We further classify internal oracles (both generic and system-specific) into three classes: data-based oracles, control-based oracles and hybrid oracles.

Test execution with respect to a data-based oracle involves checking the state of a data variable such as a variable representing a buffer. For example, the generic internal oracle for method $strcpy$ in Figure 2 (line `a`) specifies that the length of the string $strs$ stored in the buffer must be less than or equal to the size of the space $strt$ that was allocated for $strs$. The data state of $strt$ is checked whenever buffer-sensitive operations are encountered. A test case targeting a data-based oracle fails if it violates the constraint on the data value.

Some faults are caused by violations along specific control paths. Control-based oracles are used to guard against such violations. For example, consider the system-specific internal oracle shown in Figure 2 (line `c`). This oracle requires that when API ABB_db1_create is called, ABB_db1_close must also be called. A test case targeting this control-based oracle fails if ABB_db1_close is missing along any control path.

In many cases, internal oracles involve combinations of data-based and control-based oracles, and we refer to such oracles as "hybrid oracles". For instance, the generic oracle shown in Figure 2 (line `b`) requires that the semaphore acquired using *sem_acq* with object o should eventually be released by a call to *sem_rel* with object o. This requires us to check the value of the semaphore object o while also tracking the control flow related to the *sem_acq* and *sem_rel* functions.

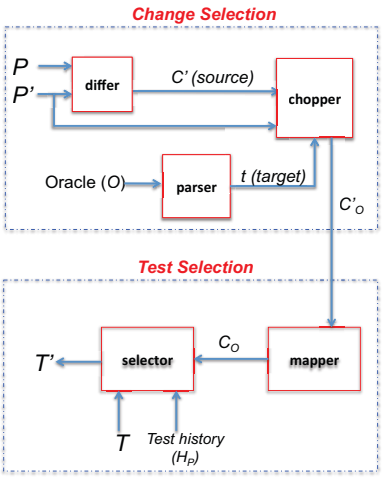

Fig. 3. Overview of RTSO

B. Overview of RTSO

The RTSO technique (Figure 3) involves a change selection component and a test selection component. Let P be the original program and let P' be a modified version of P. Let C' be the set of changes made to P to produce P' (a set of program locations in P') and let O be the set of internal test oracles (O need exist only in P'). The change selection component identifies C'_O, a subset of C' related to O. Based on set C'_O, the test selection component selects $T' \in T$ for regression testing P'.

C. Change Selection

The change selection component of RTSO computes changes that are relevant to a given set of internal oracles. This component has three modules: *differ*, *parser*, and *chopper*. Figure 4 displays the algorithm used by this component. The algorithm takes prior program version P, modified program version P', and set of oracles O as inputs, and it computes the changes that are relevant to O, denoted by C'_O. First, the algorithm invokes the *differ* module, which uses a differencing tool to identify the set of changes (C') between P and P' (line 5). Next, the algorithm invokes the *parser* to extract chopping targets (t) from internal oracle set O (line 6), where O contains one or more oracles. For each change $c' \in C'$, the algorithm invokes the *chopper* to compute impact set IS_O (lines 7-9) with respect to O relative to program P', where c' contains the chopping source. If IS_O is not empty, change c' potentially impacts one or more oracles in O, so the algorithm adds c' to the relevant change set C'_O (line 11).

In the example shown in Figure 1, statements S_7 and S_9 are involved in two changes made to produce P' from P. The *differ* module outputs $C' = \{S_7, S_9\}$, which forms the chopping source. Consider a generic data-based internal oracle, the memory security oracle, specified in Figure 2 (line `a`). The buffer-sensitive operation related to this oracle appears at statement S_{16}, where the oracle enforces the constraint that the length of string $strs$ stored in the buffer must be less than or equal to the size of the space allocated for it. Hence, the two

```
procedure ChangeSelection
1: Inputs: P, P', O
2: Outputs: C'_O
3: begin
4: C'_O = φ
5: C' = differ(P, P')
6: t = parser(O)
7: for each c' ∈ C'
8:       IS_O = φ
9:       IS_O = chopper (P', c', t)
10:       if IS_O ≠ φ
11:           C'_O = C'_O ∪ c'
12:      endif
13: endfor
14: end
```

Fig. 4. Change selection algorithm

```
procedure TestSelection
15: Inputs: T, H_P, C'_O
16: Outputs: T'
17: begin
18: T' = φ
19: C_O = mapper(C'_O)
20: for each c_O ∈ C_O
21:       for each tc ∈ T
22:           if H_P(tc, c_O) = true /* tc traverses c_O */
23:               T' = T' ∪ tc
24:           endif
25:       endfor
26: endfor
27: end
```

Fig. 5. Test selection algorithm

parameters in *strcpy* form the chopping target (t) extracted by the *parser*. The *chopper* computes $\{S_9, S_{10}, S_{16}\}$ as the impact set with respect to the given memory security oracle. As a result, $C'_O = \{S_9\}$.

D. Test Selection

The test selection component of RTSO has two modules: mapper and selector. Its algorithm is shown in Figure 5. The test selection component takes three inputs: the original test suite T, the oracle-relevant change set C'_O obtained from the change selection component, and the test coverage history H_P that denotes which test cases in T covered which statements in P, obtained earlier in the maintenance cycle by running T on P. C'_O is input to *mapper*, which returns C_O, the changes corresponding to C'_O in P' but located in the original program P (line 19). For each change $c_O \in C_O$, based on the coverage history H_P, the algorithm selects all test cases from T that traversed c_O, and adds them to T' (lines 21-24).

In the example shown in Figure 1, $C_O = C'_O = S_9$ is returned by the *mapper*. Suppose there are two test cases in T, $tc_1 = (0, aaaaaaaaa)$, and $tc_2 = (2, aaaaaaaaa)$, each with input string length nine. In this case the *selector* selects only tc_1 for T' because tc_2 does not cover anything in C_O. As a result, tc_1 causes the buffer overflow to occur at S_{16} but tc_2 does not. In contrast, most traditional RTS techniques would select both tc_1 and tc_2 for T' because they both cover changed statements $C' = \{S_7, S_9\}$.

IV. Empirical Study

In this study, we evaluate our oracle-based regression test selection technique (RTSO) in terms of *efficiency* and *effectiveness*. Efficiency measures the extent to which RTSO can reduce the cost of regression testing, while effectiveness measures the technique's ability to detect faults. We compare RTSO to the retest-all technique and to a traditional code-differencing-based RTS technique [27].

We consider the following research questions:

RQ1: How does the *efficiency* of RTSO compare to that of traditional RTS and retest-all techniques?

RQ2: How does the *effectiveness* of RTSO compare to that of traditional RTS and retest-all techniques?

A. Objects of Analysis

As objects of analysis, we chose seven programs written in C/C++. These included two open source programs downloaded from the Software-artifact Infrastructure Repository (SIR) [28], and five programs from a large industrial system developed at ABB. We utilized three versions of each of the two open-source programs and two versions of each of the ABB programs. Table I lists these program versions along with some of their characteristics, including the number of lines of non-comment code (column 2) and the number of test cases provided (column 3) with their base versions (other columns are described later).

The first open source program, MAKE, is a popular GNU utility used to control the generation of executables from a program's source files. We chose three consecutive versions of this program (v3.78.1, v3.79, v3.79.1), along with the test suite provided with the program in SIR. The second open source program, GREP, is another popular GNU utility used to search for text matching a regular expression. We chose three consecutive versions of this program (v2.3, v2.4, v2.4.1) and its SIR test suite. The five ABB programs represent five different components of the industrial system, and were already equipped with test suites. For purposes of anonymization, we denote these by the names ABB.X, ABB.Y, ABB.Z, ABB.M, and ABB.N. We chose two versions for each of these programs (denoted by v and v′). In total, then, this gave us nine pairs of programs to which RTS techniques can be applied: two of MAKE ((v3.78.1, v3.79) and (v3.79, v3.79.1)), two of GREP ((v2.3, v2.4) and (v2.4, v2.4.1)), and five of the ABB programs ($ABB.C(v), ABB.C(v')$), for $C \in \{X, Y, Z, M, N\}$.

We implemented generic and system-specific oracles such as those presented in Section III-A. We considered 15 generic oracles for MAKE and GREP, and 13 generic and 11 system-specific oracles for the ABB programs. Column 4 of Table I lists the numbers of oracles that were ultimately applicable per program. For the ABB programs, the numbers within parentheses represent the number of generic oracles.

Because engineers are typically concerned with classes of faults rather than individual oracles, we chose to study oracles in terms of fault classes. For example, more than one oracle can be used to detect faults involving buffer security, so

TABLE I
OBJECT PROGRAMS AND THEIR CHARACTERISTICS

Program	NLOC	tests	oracles	classes	faults
MAKE (V3.78.1)	17584	1046	-	-	
MAKE (V3.79)	23124	1046	14	3	37
MAKE (V3.79.1)	23257	-	14	3	26
GREP (V2.3)	9943	809	-	-	
GREP (V2.4)	10032	809	15	3	16
GREP (V2.4.1)	10073	-	15	3	13
ABB.X (V)	1747	403	-	-	
ABB.X (V')	1906	-	12(9)	4(2)	11
ABB.Y (V)	394	222	-	-	
ABB.Y (V')	494	-	12(9)	2(1)	11
ABB.Z (V)	3208	1065	-	-	
ABB.Z (V')	4826	-	14(5)	6(2)	14
ABB.M (V)	6549	456	-	-	
ABB.M (V')	6739		13(6)	4(1)	26
ABB.N (V)	5723	1443	-	-	
ABB.N (V')	6132	-	14(7)	4(1)	32

we considered a set of oracles relevant to that fault class. Specifically, we group generic oracles into the classes *buffer security*, *dynamic memory management*, *file management*, and *critical sections*. System-specific oracles are classified in terms of faults being checked for particular system features. For example, three system-specific oracles used to check database operations are classified as *database management* oracles. Column 5 of Table I lists the number of oracle classes considered for each program (the numbers within parentheses represent the number of classes of generic oracles only).

To address our research questions we also required faulty versions of our object programs. Here, we are specifically interested in *faults that are of the classes detectable by our oracles*, and we require a number of oracle-related faults adequate to allow us to study the effectiveness of our approach. Because our object programs conceivably could have contained faults initially, we first ran all of the test cases supplied with each original program on its modified version, with respect to both internal and output-based oracles. A few oracle-related faults were present in the modified programs. Specifically, four and three memory faults were detected in MAKE v3.79 and v3.79.1, respectively. Only two of these faults, however, qualified for our study because we are interested only in regression faults (i.e., faults located in changed code), and the other five faults were residual faults not located in changed code.

Two faults were not sufficient for our evaluation, so we next hand seeded additional faults of the fault classes. For each program, we first identified program changes using approaches described in Section IV-C. We next identified the changed statements for which a given fault class is applicable, and seeded potential faults related to that fault class in those statements. For data-based oracles, if a change was made to a variable V specified in the oracle, we changed the content of V or the operators that affect the computation of V; otherwise, we changed the variables having data dependencies with V or their computation operators. For example, for faults involving buffer management, we changed the length of variables of source or target strings in those buffer sensitive operations, or changed the length of variables that have data dependencies with the source or target string. For control-based oracles, if a change was made to an API defined in the oracles, we omitted that API; otherwise, we changed the conditional statements affecting its reachability. For example, for faults involving $A() \Rightarrow B()$ (i.e., if $A()$ is called, $B()$ must eventually be called), we omitted $B()$, or changed conditional statements affecting reachability of $B()$. For hybrid oracles, we used the seeding methods for both data-based and control-based oracles. For example, for faults involving critical section protection, we omitted statements involving critical section entry or exit, changed associated lock objects, or changed the variables having data dependencies with the objects.

After running the original test suites on the modified programs with seeded faults, we eliminated potential faults that could not be detected by any test case using both internal or output-based oracles. These potential faults, which can include "equivalent mutants", provide no insight into the fault detection effectiveness of the techniques. The numbers of faults ultimately considered for our object programs are reported in the rightmost column of Table I.

B. Variables and Measures

1) Independent Variable: Our independent variable involves RTS techniques. In addition to our oracle-directed RTS technique (RTSO), we consider a traditional RTS technique (RTSC) and the retest-all technique (RTA). With the latter two techniques, we employ both output-based and internal oracles. Comparing RTSO to control techniques that employ output-based oracles is necessary in order to assess whether RTSO detects faults more effectively that those techniques in their typical manner of employment. Comparing RTSO to control techniques employing internal oracles is necessary in order to determine whether RTSO is able to detect all of the faults that those techniques could detect using those oracles. As noted above, we implemented internal oracles based on the generic rules and system-specific rules presented in Section III. We implemented output-based oracles by providing output checkers appropriate to the programs.

We denote the five techniques utilized as follows:

RTSO: oracle-directed RTS;

RTSC$_P$: traditional differencing-based RTS with output checking;

RTA$_P$: retest-all with output checking;

RTSC$_I$: traditional differencing-based RTS with internal oracle checking;

RTA$_I$: retest-all with internal oracle checking.

Because our study considers only faults of the classes detectable by internal oracles, and our RTSO technique is created for checking internal oracles, we do not consider output checking in RTSO.

2) Dependent Variables: As dependent variables, as mentioned above, we measure *efficiency* and *effectiveness*. To measure *efficiency* we use two metrics. The first metric considers the numbers of test cases that are selected using the various techniques, providing a view of efficiency that is independent

of analysis technique implementation issues. To use this metric we applied RTSO, $RTSC_P$, and $RTSC_I$ and recorded the numbers of test cases selected by the approaches; we report the results as percentages of the original test suites and refer to this metric as *percentage of test cases selected*.

As a second efficiency metric we measured *testing time*, by summing three measures: the time required for code analysis (if any), the time required to execute test cases, and the time required to perform oracle or output checking activities.[1] To obtain these times, when we applied RTSO, $RTSC_P$, and $RTSC_I$, we measured the analysis time required for selection. Then, we executed each object program ten times on the various sets of selected test cases (or in the case of RTA_P and RTA_I, on all test cases), and calculated the average time required to execute the test cases and perform all oracle-based checking.

To measure fault detection effectiveness, we also use two metrics. First, to compare the fault detection effectiveness of RTSO to the fault detection effectiveness of other techniques when they are used in their typical manner (with output oracles and targeting all faults), we compare the numbers of faults detected by test cases selected by RTA_P, $RTSC_P$, and RTSO, respectively. We use the term *fault detection performance* to denote this comparison metric.

Second, to compare the fault detection effectiveness of RTSO to the fault detection effectiveness that other techniques would produce if they were to target the fault classes targeted by internal oracles, we compare the number of faults detected by test cases that are selected by RTSO, RTA_I, and $RTSC_I$, respectively. We use the term *fault detection inclusiveness* to denote this comparison metric.

C. Study Operation

For our *differ* module, we used two differencing tools to perform change identification for successive versions of our object programs. The first tool is an existing ABB internal tool tailored for ABB programs, and was implemented based on an *Abstract Syntax Tree* (AST) differencing technique [29]. This tool did not directly work on our open-source objects, MAKE and GREP, so for these programs we implemented a second tool based on textual differences in accordance with an existing differencing algorithm [27].

We implemented our *chopper* module using the program chopping APIs provided by *CodeSurfer* [30]. The chopper module takes program changes and internal test oracles as inputs, and outputs change impact sets relevant to the oracles. The accuracy of program chopping in CodeSurfer depends on several configurable static analysis parameters such as the algorithm used for pointer analysis, context/flow sensitivity, and non-local analysis. In our study, we use the most "expensive" options available with CodeSurfer for these static analysis parameters. Ideally, we should use both data dependencies and control dependencies when using program chopping; however, this may yield large impact sets, causing many more test cases to be selected. Thus, in our study, we used only data dependencies for all classes of oracles. (For the objects studied this had no impact on fault detection effectiveness; however, this may not always be the case).

We implemented our internal oracles using binary instrumentation. Binary instrumentation tools have been widely used to detect errors in software. We chose two popular tools: PIN [31] and Valgrind [32]. Valgrind can be directly used to detect memory faults such as buffer overflow and memory leaks. PIN enables us to create customized dynamic monitoring modules to monitor violations of different classes of oracles. All ABB dynamic rule checkers were implemented using PIN, and the dynamic rule checkers for MAKE and GREP used both PIN and Valgrind. To implement output oracles for MAKE and GREP, we used the Linux utilities *diff* and *cmp* to compare actual outputs of test cases executed on original and modified programs. For the ABB programs, output oracles had already been implemented as part of the test cases, and program crashes (e.g., segmentation faults) are also monitored.

To determine which test cases detect which faults, we executed all test suites on the faulty versions of each object program with only one fault activated per execution to avoid interactions between faults. (This approach may overestimate fault detection in cases where multiple faults would actually *mask* each other's effects, but facilitates evaluation, and in prior work [33] has been shown to have negligible effects on results.) When each fault is activated, all oracles that pertain to that fault class become active and related program behaviors are monitored. All oracle checking (including output checking) is done at runtime. This approach helps us expedite the experimental process without affecting the results. In practice, of course, developers may not know the fault types ahead of time, and would enable all oracles regardless of fault types.

D. Threats to Validity

The primary threat to external validity for this study involves the representativeness of our programs, faults, and test cases. Other systems may exhibit different behaviors and cost-benefit tradeoffs, as may other forms of test cases. However, we do reduce this threat to some extent by using both open source and industrial objects for our study. A second threat to validity involves the baseline techniques that we compare against, which include traditional RTS and retest-all techniques. Other baseline techniques may also be of interest.

The primary threat to internal validity for this study is possible faults in the implementation of our approach and in the tools that we use to perform evaluation. We controlled for this threat by extensively testing our tools and verifying their results against a smaller program for which we can manually determine the correct results. We also chose to use popular and established tools (e.g., CodeSurfer, Valgrind and PIN) to implement the various modules in our approach. Finally, the

[1] When measuring testing time, we do not include time spent on activities performed in the preliminary phase of regression testing (prior to the time at which the modified program is available for testing, and when testing time becomes a critical issue); these include collection of test history information, construction of the system dependence graph for the original program, and instantiation of test oracles.

differencing tool used at ABB has been extensively used by industrial developers.

Where construct validity is concerned, our measurements of efficiency focus on the time required for analysis, test execution, and application of oracles. However, other costs such as test setup and maintenance costs can also play a role in regression testing efficiency. Our measurements of fault detection effectiveness are relative to specific classes of faults. Our technique may not detect other classes of faults beyond those related to the specific oracles used; however, it is intended to apply only to specific fault classes.

E. Results and Analysis

1) RQ1: Technique Efficiency: To address our first research question we begin by considering percentages of test cases selected. Figure 6 displays these results for the RTSC and RTSO selection techniques, for each of the nine object programs. Internal and output-based oracles lead to identical test selection percentages given our objects and test cases; thus, we do not differentiate between oracles in this case. As the results show, RTSC selected smaller test suites than RTA in some cases, but the overall savings were not dramatic; the percentage of selected test cases ranged from 71.2% to 100%. RTSO, on the other hand, selected smaller test suites on all nine programs (in most cases dramatically), with selection percentages ranging from 2.9% to 89.7%. In fact, on seven of the nine versions, RTSO selected fewer than 60% of the test cases, and in two cases (MAKE2 and ABB.X) it selected fewer than 10% of the test cases.

We now consider savings in terms of testing time, and here we consider all five techniques: RTA_P, RTSC_P, RTA_I, RTSC_I and RTSO. Figure 7 shows testing time measured in minutes for each of the nine object programs, with each of the five techniques. Among the three techniques using internal oracles (the rightmost three bars in each set of five), RTSC_I required less testing time than RTA_I on eight of the nine programs programs (all but ABB.Y). The savings for individual programs ranged from 1.2% to 34%, and the average savings across all nine programs was 24.5%. The savings on the open source programs averaged 6.9%, and on the ABB programs it averaged 30.2%. RTSO achieved even greater savings than RTA_I, with an average savings across all programs of 68.5%, and savings on individual programs ranging from 8.6% to 97.3%. Savings on the open source programs averaged 62.7%, and on the ABB programs it averaged 70.4%.

When compared to techniques using output-based oracles (the two leftmost bars in each set of five), RTSC_I did not result in savings in testing time. This is because with binary instrumentation, internal oracle checking caused test execution time to increase by 1.5 – 5 times across the nine programs.

To determine what portion of total testing time is spent performing analysis tasks, we also report just the analysis time required for RTSC_P, RTSC_I and RTSO. Table II shows the percentage of total testing time spent on analysis for each program using the three techniques. In all but three cases, analysis time accounted for less than 5% of overall testing

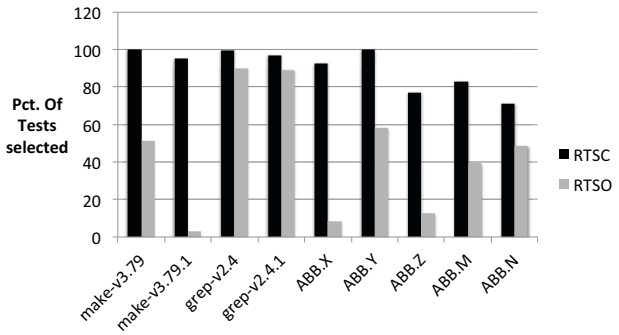

Fig. 6. Overall efficiency: percentages of test cases selected

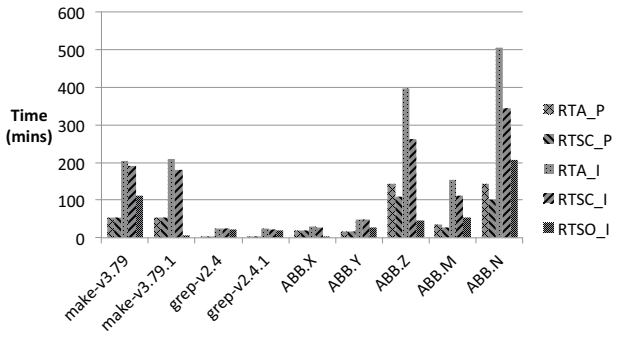

Fig. 7. Overall efficiency: testing time

time. In quantitative terms, the time spent on the analysis performed by test case selection never exceeded 77 seconds, and the time spent on oracle-based impact analysis never exceeded 16 seconds. As a result, we find this cost to be negligible. Thus, on systems for which the time required to execute test cases is larger than those we consider in this study, the reduction in numbers of test cases selected for the oracle-based techniques may translate more readily to savings in test execution time.

2) RQ2: Technique Effectiveness: To address our second research question, we first consider fault detection performance, by examining whether RTSO achieved higher fault detection effectiveness results than RTA_P and RTSC_P. Table III shows the numbers of faults detected by the test cases selected by all RTS techniques for each of the nine programs. Recall again that the faults we utilized are all of the classes of faults targeted by our internal oracles, and the total numbers of faults are shown in the last column of Table I. In all cases, RTSO detected all faults, and overall, improved fault detection effectiveness by 250.9% across all nine programs compared to RTA_P and RTSC_P.

We next consider fault detection inclusiveness; that is, whether faults detected by RTA_I and RTSC_I can also be detected by RTSO. Results in Table III (second row) show that RTSO achieved 100% inclusiveness; that is, in the instances considered in this study, RTSO selected test cases that, together, revealed all faults in the programs. Note that this result was not fore-ordained, because RTSO is not safe when control dependence information is not considered; however,

TABLE II
PERCENTAGE OF TIME SPENT ON ANALYSIS (%)

Technique	make-v3.79	make-v3.79.1	grep-v2.4	grep-v2.4.1	ABB.X	ABB.Y	ABB.Z	ABB.M	ABB.N
RTSC$_P$	2.4	2.3	35.4	36.9	1.1	1.1	0.8	1.4	0.5
RTSC$_I$	0.7	0.7	3	3.3	0.7	0.4	0.3	0.4	0.1
RTSO	1.4	25.7	3.7	4.3	8.5	1.0	2.2	0.9	0.3

TABLE III
NUMBERS OF FAULTS DETECTED

Technique	make-v3.79	make-v3.79.1	grep-v2.4	grep-v2.4.1	ABB.X	ABB.Y	ABB.Z	ABB.M	ABB.N
RTA$_I$/RTSC$_I$/RTSO	37	26	16	13	11	11	14	26	32
RTA$_P$/RTSC$_P$	10	6	7	8	3	6	8	3	2

the result does show that inclusiveness need not necessarily be sacrificed by our approach. To understand this result further, we examined each faulty version for each program, and found that all faulty program changes were identified as relevant changes by our change selection component; this explains why RTSO could detect all faults in these cases.

V. DISCUSSION

The foregoing results have two primary implications for the use of RTS techniques:

- Our oracle-based RTS technique is more expensive in terms of testing time than retest-all and code-differencing based RTS techniques when those techniques use output-based oracles; however, our technique has substantially better fault detection ability than those techniques relative to the classes of faults it targets.
- If we attempt to employ the same internal oracles in the retest-all and code-differencing based RTS techniques that are used in our oracle-based technique, we can detect the same sets of faults with all techniques, but our oracle-based technique is substantially more efficient than the other techniques.

In other words, if these results generalize to other programs and RTS techniques, then *if engineers wish to target the classes of faults that our oracles target, the oracle-based technique is the best technique to utilize.*

We now explore additional observations relevant to our study, and factors that cause performance differences.

First, the percentage of test cases RTSO selected varied across programs. Test case selection results depend on the program locations in which changes occur and where oracles are applied; both of these factors affect the results of program chopping, and thus affect the oracle-relevant changes identified for test selection. For the two versions of GREP, on which RTSO selected relatively large sets of test cases, many changes occur inside the *main* functions, and all of these are identified as relevant to the oracles; thus, a relatively large percentage of test cases were selected.

Where testing time is concerned, RTSO produced savings on just three out of nine programs, compared to RTA$_P$ and RTSC$_P$. There are two reasons for this. First, testing with internal oracles can be expensive because instrumentation can dramatically slow down the program. Second, testing can be even more expensive when we use multiple instrumentation techniques to check different oracles, requiring us to run test suites more than once, as we did for our open source programs; a more efficient instrumentation technique could reduce such costs. Again, however, these added costs do allow the oracle-based approach to outperform the other (output-based) approaches in terms of fault detection effectiveness.

The RTS technique that we have presented is intended ultimately for use on larger systems than those we have studied. The code portions of the ABB programs we studied constitute no more than 1% of the entire ABB system. In fact, running all of the test cases on the full system requires weeks of effort. It is worth commenting on how our results might scale to larger systems. A primary constituent of testing time when our internal oracles are utilized involves the cost of code instrumentation. The fewer the test cases required by an RTS technique, the lower the instrumentation costs become. Compared to analysis time (0.3% to 25.7%), instrumentation overhead (1.5 to 5 times the cost of test execution) is more expensive. We conjecture that for larger systems with longer test execution times, the possibilities for achieving substantial savings in testing time will increase.

In our study, we restricted our program chopping approach to consider data dependencies only. Although this approach may impact test selection results in general, in our particular study it did not affect the inclusiveness of our technique. The reason for this is that for each program version, there are no changes such that the faults injected could cause control flow to be altered in a manner that affects reachability of the function in the oracle (e.g., function $B()$ in $A(o) \Rightarrow B(o)$).

Although RTSO is effective for detecting faults related to internal oracles, it is possible that internal oracles could produce false positives. In such cases, an oracle might suggest that a fault exists when in fact there is no fault, and if such false positives were to occur at a certain level of frequency, this could increase testing costs as engineers spend time investigating fault reports. However, no false positives were found in our study. Additional empirical work will be needed to investigate this issue further.

Finally, we consider fault detection results relative to the individual classes of faults detected by our oracles. Figure 8 shows, for each of the nine versions of our seven object programs, the percentages of test cases selected by RTSC (first bar), RTSO with all oracles (second bar), and RTSO with individual oracle classes (remaining bars labeled O.I,

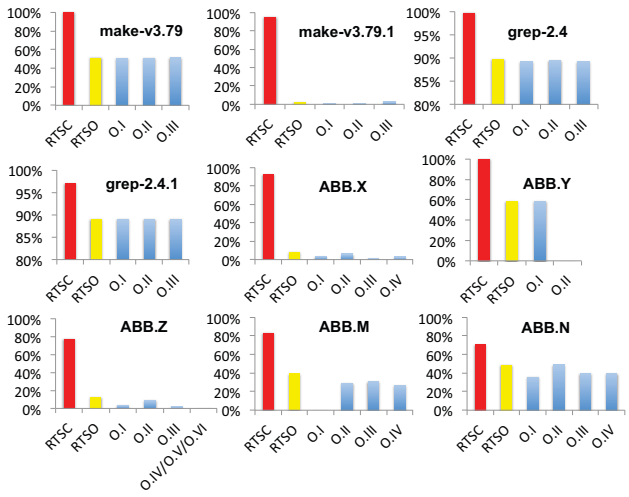

Fig. 8. Percentages of test cases selected for specific oracle classes

O.II, etc), over the set of modified versions of that program. There are several cases in which no test cases are selected for a particular oracle class (i.e., O.II of ABB.Y; O.IV, O.V, and O.VI of ABB.Z; O.I of ABB.M), which indicates that no regression test selection is needed for these oracle-related faults. For all nine programs, the test cases selected for individual oracle classes formed overlapping sets. In fact, the overlap between test cases was particularly large for all but two programs (ABB.X and ABB.Z).

There are two reasons for these results. First, program changes are likely to be concentrated in one function or block, causing selected relevant changes to be concentrated too; thus, test cases traversing the changed statements are unlikely to be disjoint. Such cases occur in the ABB programs when a new feature is added to the modified program in the form of new functions, or when entire features (functions) are changed. Second, one change may affect more than one oracle class, causing the relevant changes computed for each oracle class to overlap. For example, suppose a statement involving memory allocation is changed, and this allocated memory is later used to perform buffer writing. Suppose that the string written into the buffer is a file name that is passed as a parameter into a file open operation. In this case, the changed statement is relevant to three oracle classes: *buffer security*, *dynamic memory management*, and *file management*.

VI. Related Work

There has been a great deal of research on improving regression testing through regression test selection, test case prioritization, and test suite reduction; Yoo and Harman [1] provide a recent survey. Here, we restrict our attention to techniques that share similarities with ours.

Static program slicing has been used for regression testing [16], [17]. Bates et al. [16] apply program slicing to both old and modified versions of a program to identify statements having *equivalent execution patterns*. Statements are considered "affected" if they are not equivalent, and test cases associated with these affected statements are selected. Binkley [17] extends Bates et al.'s technique to interprocedural regression test selection. Our technique, like these, uses slicing, but its use of test oracles is novel.

Some RTS techniques target specific faults. Leung et al. [34] introduce a *firewall* technique that selects integration tests based on module changes. Robinson et al. [35] apply *firewall*-based techniques for regression testing industrial real-time systems. Similar to our technique, firewall-based techniques select tests that target particular fault classes; however, they do not select test cases related to internal oracles.

There has been some work on performing test case prioritization using oracles. Jeffrey et al. [9] perform prioritization based on the number of program statements relevant to print statements, an approach that relies on output oracles. Staats et al. [10] present a prioritization technique that utilizes information on coverage of values that have impact on variables used in internal, assertion-based test oracles. Both of these techniques, like ours, ultimately reward test cases whose executions influence a part of the program that can produce verifiable testing outputs. However, we focus on regression test selection, not prioritization. Still, the techniques used in these papers to reward test cases could also be adapted to regression test selection (and vice-versa).

Zheng et al. [36] propose an RTS technique based on dynamically mined operational models. These models (rules) are used as internal test oracles in regression testing, and only test cases causing rule violations are selected. Zheng's technique and ours are similar in that both select test cases targeting test oracles. However, the oracles we consider are commonly used generic rules and real system rules that can be used across multiple program versions, whereas the oracles mined by Zheng's technique are predicate rules that might not be applicable for other program versions. Also, Zheng's technique does not restrict test selection to test cases that are traverse changes, rendering it less precise, and it selects all test cases that cause rule violations in the old programs. Our technique, in contrast, selects only test cases traversing changes relevant to the oracles in the modified programs.

Fault-based testing [37], which focuses on certain restricted classes of well-formalized faults, has been used for regression test selection [38], [39], [40], [41]. Similar to our technique, fault-based testing techniques select test cases that detect particular types of faults. For example, Chen et al. [38] and Tai et al. [41] use boolean functions to express faults involving misuse of literals in boolean operators, and select test cases detecting these faults. In practice, however, it is impractical to consider each boolean operator fault in real regression testing scenarios such as those involving large industrial software systems. Also, these techniques do not use test oracles for a particular fault class that may include varieties of single faulty statements (e.g., operator faults, missing statements, etc). In contrast, our technique uses oracles that are practical, and that are able to detect more than just single fault types that may propagate to the oracles.

Specification-based RTS techniques use system requirements to select test cases [42], [43]. However, these techniques do not consider the effectiveness of using internal oracles for particular fault classes. For example, Chittimalli et al. [42] present a technique to select test cases associated with system requirements. Although requirements can be used as oracles, such techniques focus on "selection" while ignoring these oracles. Also, our technique uses program analysis to identify changes that could potentially propagate to the oracles, whereas Chittimalli's considers only changes within the code related to requirements and hence may omit test cases.

VII. Conclusion

We have presented an RTS technique that uses internal oracles. We have conducted an empirical study applying our technique to two open source programs and five components of a large ABB system. Our results show that our technique can be more effective and efficient than other RTS techniques when targeting classes of faults related to internal oracles.

As part of our future work, we intend to extend our study to consider more components of the ABB system with additional oracle classes. We also intend to investigate how data dependence and control dependence information used in program chopping can affect change (and hence test case) selection for data-based, control-based and hybrid oracles. Finally, we will perform more extensive empirical studies.

VIII. Acknowledgements

This work has been partially supported by the National Science Foundation through award CNS-0720757, and the Air Force Office of Scientific Research through award FA9550-10-1-0406, and by the World Class University program under the National Research Foundation of Korea and funded by the Ministry of Education, Science and Technology of Korea (Project No: R31-30007).

References

[1] S. Yoo and M. Harman, "Regression testing minimisation, selection and prioritisation: A survey," *J. Softw. Test. Verif. Rel.*, vol. 22, no. 2, 2012.
[2] G. Rothermel and M. J. Harrold, "Empirical studies of a safe regression test selection technique," *IEEE Trans. Softw. Eng.*, vol. 24, no. 6, 1998.
[3] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel, "An empirical study of regression test selection techniques," *ACM Trans. Softw. Eng. Meth..*, vol. 10, no. 2, 2001.
[4] M. Acharya and B. Robinson, "Practical change impact analysis based on static program slicing for industrial software systems," in *Int'l. Conf. Softw. Eng.*, 2011.
[5] M. Staats, M. Whalen, and M. Heimdahl, "Programs, tests, and oracles: The foundations of testing revisited," in *Int'l Conf. Softw. Eng.*, 2011.
[6] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," in *Int'l Symp. Softw. Test. Anal.*, 2010.
[7] M. Staats, G. Gay, and M. Heimdahl, "Automated oracle creation support, or: How I learned to stop worrying about fault propagation and love mutation testing," in *Int'l Conf. Softw. Eng.*, 2012.
[8] R.-G. Xu, P. Godefroid, and R. Majumdar, "Testing for buffer overflows with length abstraction," in *Int'l. Symp. Softw. Test. Anal.*, 2008.
[9] D. Jeffrey and N. Gupta, "Test case prioritization using relevant slices," in *Int'l. Comp. Softw. Appl. Conf.*, 2006.
[10] M. Staats, P. Loyola, and G. Rothermel, "Oracle-centric test case prioritization," in *Int'l Symp. on Softw. Rel. Eng.*, 2012.
[11] K. Poulsen, "Tracking the blackout bug," *SecurityFocus*, 2004.
[12] G. Xu, M. D. Bond, F. Qin, and A. Rountev, "LeakChaser: Helping programmers narrow down causes of memory leaks," in *Prog. Lang. Des. Impl.*, 2011.
[13] S. Park, S. Lu, and Y. Zhou, "CTrigger: Exposing atomicity violation bugs from their hiding places," in *Arch. Sup. Prog. Lang. Op. Sys.*, 2009.
[14] T. Yu, A. Sung, W. Srisa-an, and G. Rothermel, "Using property-based oracles when testing embedded system applications," in *Int'l Conf. Soft. Test. Verif. Val.*, 2011.
[15] B. Sun, G. Shu, A. Podgurski, and B. Robinson, "Extending static analysis by mining project-specific rules," in *Int'l. Conf. Softw. Eng.*, 2012.
[16] S. Bates and S. Horwitz, "Incremental program testing using program dependence graphs," in *Princ. Prog. Langs.*, 1993.
[17] D. Binkley, "Semantics guided regression test cost reduction," *IEEE Trans. Softw. Eng.*, vol. 23, no. 8, 1997.
[18] J. Voas and K. Miller, "Putting assertions in their place," in *Intl. Symp. Softw. Rel. Eng..*, 1994.
[19] M. Staats, M. W. Whalen, and M. P. Heimdahl, "Programs, tests, and oracles: The foundations of testing revisited," in *Int'l. Conf. Softw. Eng.*, 2011.
[20] T. Yu, W. Srisa-an, and G. Rothermel, "SimTester: A controllable and observable testing framework for embedded systems," in *Conf. Virtual Exe. Envs.*, 2012.
[21] R.-Y. Chang, A. Podgurski, and J. Yang, "Discovering neglected conditions in software by mining dependence graphs," *IEEE Trans. Softw. Eng.*, vol. 34, no. 5, 2008.
[22] N. Robertson and P. D. Seymour, "Graph minors. I. Excluding a forest," *J. Comb. Theory, Ser. B*, vol. 35, no. 1, 1983.
[23] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," in *Prog. Lang. Des. Impl.*, 1988.
[24] Klocwork, http://www.klocwork.com/.
[25] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng, "Bandera: Extracting finite-state models from Java source code," in *Int'l. Conf. Softw. Eng.*, 2000.
[26] M. Weiser, "Program slicing," in *Int'l. Conf. Softw. Eng.*, 1981.
[27] F. I. Vokolos and P. G. Frankl, "Pythia: A regression test selection tool based on textual differencing," in *Int'l. Conf. Rel. Qual. Safety Soft.-int. Sys.*, 1997.
[28] SIR, "Software-artifact Infrastructure Repository," Web-page, http://sir.unl.edu/portal/index.php.
[29] I. Neamtiu, J. S. Foster, and M. Hicks, "Understanding source code evolution using abstract syntax tree matching," in *Intl. W. Mining Softw. Repositories*, 2005.
[30] CodeSurfer, http://www.grammatech.com/products/codesurfer/overview.html.
[31] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Prog. Lang. Des. Impl.*, 2005.
[32] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Prog. Lang. Des. Impl.*, 2007.
[33] G. Rothermel, S. Elbaum, A. G. Malishevsky, P. Kallakuri, and X. Qiu, "On test suite composition and cost-effective regression testing," *ACM Trans. Softw. Eng. Meth.*, vol. 13, no. 3, Jul. 2004.
[34] H. K. N. Leung and L. White, "Insights into testing and regression testing global variables," *J. Softw. Maint.*, vol. 2, no. 4, 1990.
[35] L. White and B. Robinson, "Industrial real-time regression testing and analysis using firewalls," in *Int'l. Conf. Softw. Maint.*, 2004.
[36] W. Zheng, M. Lyu, and T. Xie, "Test selection for result inspection via mining predicate rules," in *Int'l Conf. Softw. Eng. – NIER Track*, 2009.
[37] L. J. Morell, "A theory of fault-based testing," *IEEE Trans. Softw. Eng.*, vol. 16, no. 8, Aug. 1990.
[38] T. Y. Chen and M. F. Lau, "Test case selection strategies based on boolean specifications," *J. Softw. Test. Verif. Rel.*, vol. 11, no. 3, 2001.
[39] K. A. Foster, "Error sensitive test cases analysis (ESTCA)," *IEEE Trans. Softw. Eng.*, vol. 6, no. 3, 1980.
[40] C. ai Sun, "A constraint-based test suite reduction method for conservative regression testing," *J. Softw.*, vol. 6, no. 2, 2011.
[41] K.-C. Tai, "Theory of fault-based predicate testing for computer programs," *IEEE Trans. Softw. Eng.*, vol. 22, no. 8, Aug. 1996.
[42] P. K. Chittimalli and M. J. Harrold, "Regression test selection on system requirements," in *India Softw. Eng. Conf.*, 2008.
[43] R. Paul, L. Yu, W.-T. Tsai, and X. Bai, "Scenario-based functional regression testing," in *Int'l. Comp. Softw. Appl. Conf.*, 2001.