Context-Aware Regression Testing

Abstract—**Regression testing is an essential process for maintaining the quality of evolving software. Existing techniques focus on the impact of code changes on the program behavior. However, changes in the execution context, such as in the library or in the database can also affect the behavior of the software. This paper presents a new regression test selection approach that not only accounts for modification of the code, but also for the changes in the execution context. Context-Aware Regression Testing uses the states of the pre-and postconditions of a function to determine if the function is affected by the change, and selects tests that execute the affected functions. The pre- and postconditions are denoted by likely invariants that preserve the properties required for, or resulting from, correct execution of the function. After a change, if the pre-and/or postconditions of a function are affected, then the tests executing this function will be selected for regression testing. Our empirical studies show that the Context-Aware Regression Testing effectively selected all the fault-revealing test cases, which suggests that the technique is safe and effective.**

Regression Testing

I. Introduction

Most software systems are continuously evolving to better serve users needs. Consequently, maintenance activities need to be frequently performed on the multi-version production software. When a change is made to the software, new tests are created to validate the new and/or changed features. In addition, regression testing, which re-runs the previous successfully executed test cases on the modified software, should be performed to investigate if the change adversely introduces regression faults. Furthermore, the execution context, such as the runtime libraries, APIs, or the databases, may also be upgraded. However, when a software library or an external API, is upgraded to a new version that is not backward compatible to the ones that the application depends on, then the behavior of the application may be affected. In [21], the results of their empirical studies suggest that changes in the execution environments, such as in the libraries and APIs, most likely would affect program behavior. Library compatibility issues are often encountered in multi-team software developments, where each team uses its own execution environment that may differ from the one in the production or used by other teams, and backward or forward compatibility problems may occur. Such compatibility issues have been broadly discussed [21], but effective regression testing to resolve these issues has yet to be explored. Although there is no change to the code, the changes of the execution environment can affect the behavior of the software. Therefore, regression testing must be performed as well to check if the software works correctly in the new execution environment.

Regression testing can be time consuming, when new tests are continuously added to the regression test suite. To minimize the maintenance cost, and to prevent service disruption of production software, an effective selection from the regression test suite to perform efficient regression testing is essential. Safe selective regression testing that aims at minimizing the cost while maintaining the effect of retest-all has been well studied [4], [14], [27], [32]. The key assumption for safe controlled regression testing is that when a modified program P is tested with a test case t, all factors that might influence the output of P, except for the code in P, are kept constant with respect to their states when the original program P is tested with t. This assumption cannot be held if the execution environment is changed and can potentially affect the behavior of the modified program P. Existing safe approaches focus on selecting modification-traversing tests that execute the changed code of the program . When the execution environment that the application depends on is changed, then these techniques will not be safe.

Few studies address the impact of changes in configuration files or databases on the behavior of the software. Haraty et al. [12] presented a regression test selection techniques for SQL-based applications. It uses a control flow graph and a firewall approaches to identify components affected by changed database modules and to select tests that exercises these components. Nanda et al. [22] presented an approach to address regression test selection when there is no code change, but property files or databases are changed. They used a traceability matrix to keep track of the test cases, in which there are executed program elements affected by a changed property in the configuration file or changed database elements.

This paper presents a safe regression test selection technique, Context-Aware Regression Testing (CART), that accounts for both changes in the code and in the execution context. CART selects tests that executes functions affected by changed functions in the program, upgraded/downgraded library or APIs, changed databases or configuration files. We use program states before and after an invocation of a function to determine if the function is affected by the change and can potentially contain regression faults. A program state before a function call denoted in the precondition of a function call including the properties required for the successful execution of the function. The state after the function call is indicated in the postcondition describing the properties after the execution of the function. These properties may involve the variables (and their values) the function needs to use, the library and APIs the function needs to call, and the object entities (associated with external database entities) the function needs to access. The test cases whose execution traces include at least one program property preserved for all the successful executions but is affected by the changes are potentially fault-revealing and need to be retested.

We use program invariants as means for capturing persistent program properties for successful executions of functions. A number of studies have suggested that program invariants can be used to investigate program behavior [10], [11] . A program invariant is a condition that holds the programs property at a given point, where the invariant should hold a true value if the programs execution behaves as expected; otherwise, a false value should be observed. To automatically detect program invariants, we use the Daikon invariant detector, developed by Ernst et al. [10], to collect likely invariants from the regression test suites. Daikon has been broadly used in various areas of software engineering [24], [26], [30], [31], [20], [7], [8], [25].

A pre- or postcondition contains invariants inferred from a class/object attribute, an element defined in a property/configuration file, a function parameter, or an entity object corresponding to a database entity. In addition, we add all the callees, including library and API calls in the precondition. The change impact on the functions in the modified program can be (1) if a function is changed, then the invariants at the post-condition of the changed function or its callers will be affected, (2) an attribute or an element in the property/configuration file is changed, then all the functions whose preconditions have invariants inferred by the changed attribute or element are affected. (3) If a function makes a call to a runtime library or API call, then there will be an invariant in the functions pre-condition indicating the call. When the library or API is changed, then this corresponding invariant is affected. (4) In modern software systems, each piece of persistent data stored in the external entities is normally associated with an entity object. The precondition of a function that makes use of an entity object will include an invariant indicating the entity object, which is affected if the associated database entity is changed. For each test execution, we keep track of the invariants in the pre- and postcondition of each function call and build an Invariant Traceability Matrix (ITM). This process only needs to be performed once can be done during the testing phase. In the regression testing, we parse ITM and select tests that include at least one affected invariant.

We conducted a set of empirical studies on nine multi-version open source software with real regression faults to investigate the effectiveness of the proposed approach, and one study on an educational software to demonstrate the effect of CART for regression test selection on software that has multiple types of changes. Three approaches were used for comparison, including retest-all, CART, and an execution trace based approach. The results of the empirical studies show that our approach is as safe as the retest-all, and as precise as the execution trace based approach.

The contributions of this paper are:

1. The impact of the changes in the execution context on the program behavior is difficult to identify and resolve in software maintenance. To our best knowledge, the existing techniques for regression testing have barely addressed this issue. Most of the safe selection approaches assume that the factors that are external to the subject program remain constant, which is often impractical for modern multi-team developed evolving software. This paper presents a new approach addressing this challenging issue that can effectively select the test cases relevant to the changed environment.

2. Existing approaches tackle single type of change, CART can effectively handle different types of changes at the same time. 3. Our approach is fully automated and we have implemented a prototype to conduct the empirical studies. The results show that CART significantly reduces the size of the regression tests while selecting all the fault-revealing regression tests. Thus, regression testing can be performed efficiently and effectively to maintain the quality of the software after changes.

4. The subject programs and the regression faults used in the empirical studies are real industrial programs and real faults. These regression faults were identified along the evolution processes, which demonstrates the feasibility to be applied to the real life industrial systems. The remainder of this paper is organized as follows: the proposed approach is described in Section 2. Section 3 presents our empirical studies. Section 4 gives an overview of the related techniques. The conclusions and the future work are given in Section 5.

II. METHODOLOGY

The goal of our test selection technique is to identify test cases that were correctly executed before the change of the program or the execution context, but potentially can be affected by the change. The basic unit of the investigation in our approach is function. For function changes, we use the postcondition of a function to determine if it is affected by the change by comparing the property in the postcondition before and after the change. For data changes (class/instance attributes, elements in configuration files, parameters), we check the precondition of each function to determine if the changed data will be used by, and affect, the function. If the changes take place in the execution context, we examine the precondition of each function to identify if the changed library or data entity in the database is used in the function.

To automatically obtain pre- and postconditions, we used Daikon [10] to extract likely invariants at the entry and exit points of each function and used them to determine the pre- and postconditions of the function. The invariants we used include function parameters and return values, global variables used (read/write) in the function, class attributes used (read/write) in the function, and conditional invariants or implications, such as $\text{this.left} \neq \text{null} \Rightarrow \text{this.left.value} \leq \text{this.value}$. In addition, we include the names of the functions called by the function in the preconditions. In [5] the results of the empirical studies suggested that if a program fails in a test execution, then there exists at least one invariant produced from Daikon that is violated (holds a different value from the one in the successful executions). Therefore, if the execution of a test does not yield to any violation of the invariants, then it is likely that the test will not fail. Under this assumption, if all the invariants obtained from the modified program hold as they were in a regression test case, then the test case is likely not fault-revealing and will not need to be re-tested.

We use the following examples to demonstrate how the invariants are used to determine if a function is affected by a change. Figure 1 shows an example ShoppingCart class, its attributes: totalCost and products, and method removeProduct. The removeProduct function removes a product from the shopping cart and updates the value of the class attribute totalCost. The attribute totalCost is involved in the program invariants in the precondition and post-condition of removeProduct. The value of totalCost is greater than or equal to zero in the precondition, and after the execution of the function, it is less than the one prior to the execution. Table 1 shows the invariants in the pre- and postconditions of the method.

```
Public class ShoppingCart{
   private float totalCost;
   private List<Product> products;
    ....
   Public void removeProduct(Product p ){
        products.remove(p)
                      .
        totalCost = totalCost − p.price
   }
}
```

Figure 1. Example 1: ShoppingCart.

TABLE I
THE INVARIANTS OF REMOVEPRODUCT.

Program Point	Invariant
...	...
ShoppingCart. removeProduct: Enter	totalCost>=0
ShoppingCart. removeProduct: Exit	totalCost<orig(totalCost)

Our approach for selecting regression tests is described as follows:

(1). If an attribute (a variable) in a program P is changed in the modified program P', then all the functions that use this changed attribute (variable) are affected. The test cases exercising these functions will need to be re-tested. In the ShoppingCart example, when the name of the attribute totalCost is changed to Cost, then these invariants will be replaced by the invariants inferred by Cost. Any test case that includes a function call whose precondition involves totalCost should be re-tested.

(2). If a faulty function f is corrected, and the modified version is f'; then a new test set T' will be created to test f' until all tests in T' pass.

(2.1) If all tests in T that exercise f and their postconditions of f are different from those of f' in T', then they will need to be retested, and their postconditions will be updated.

(2.2) Every function f'' in P that is not changed in P', but has preconditions affected by the postconditions of f', is affected. All the tests that exercise f'' will be retested.

```
public class ShoppingCart{
   Private float totalCost;
   Private List<Product> products;
    ....other attributes and functions
   Public void removeProduct(Product p ){
        if(p in products){
             products.remove(p)
             totalCost −=p.price
              }
        }
     Public void checkout(){
        return totalCost;
      }
```

Figure 2. Example 2: modified removeProduct.

TABLE II
PRE- AND POSTCONDITIONS OF MODIFIED REMOVEPRODUCT.

Program Point	Invariant
...	...
ShoppingCart.remove Product:Enter	totalCost$\geq$ 0
ShoppingCart.remove Product:Exit	totalCost$\leq$ orig(totalCost)
ShoppingCart.remove Product:Exit	totalCost$\geq$ 0
ShoppingCart.checkout: Enter	totalCost has a value
ShoppingCart.checkout: Exit	totalCost has a value

Figure 2 shows the modified function removeProduct, which deletes a statement that checks if the product to be removed is in the shopping cart. The invariant totalCost $\leq$ orig(totalCost) is changed to totalCost $<$ orig(totalCost), which in turn affects the precondition of the function checkout: totalCost has a value. Therefore, every test that exercises checkout function should be retested.

(3). When a new function g is added toP', new tests will be created to test g. Some functions in P will be modified in P' to call g and the test selection with respect to these functions is the same as (2).

```
promotion_cate_1=1
Promotion_1=10%
Desc_1= 10% discount
Code_1= happy Friday
promotion_cate_2=2
Promotion_2=20%
....
```

Promotion configuration file.

```
public class ShoppingCart{
    Private float totalCost;
    Private List<Product> products;
    ....other attributes and functions
    Public int promotion(){
    PropertiesConfiguration pcfg =
        new PropertiesConfiguration
            ("config/promotion.properties");
        Return pcfg.int("promotion_1");
}
        Public void checkout(){
            return totalCost-total*promotion();
        }
}
```

Figure 3. A new function promotion is added to ShoppingCart class.

TABLE III
THE INVARIANTS AFTER THE CHANGE.

Program Point	Invariant
...	...
ShoppingCart.promotion(): Enter	totalCost is not null
ShoppingCart.remove Product:Exit	Return 10%
ShoppingCart.checkout: Enter	totalCost $\geq$ 0
ShoppingCart.checkout: Exit	totalCost=orig(totalCost*(1-10%))
PropertiesConfiguration. Configuration(): Enter	url = config/ promotion. properties
PropertiesConfiguration. Configuration(): Exit	url = config/ promotion. properties

Figure 3 shows an example where ShoppingCart has a new function promotion which reads in a promotion profile and returns the first promotion that has 10% discount for each sale. When it reads the profile, it calls propertiesConfigurations configuration function and sets the address of the profile. The parameter URL is captured in the invariant which has a constant value, config/promotion.properties. When this property file has some content changes, then every test that has this property file in the precondition of any function it exercised will need to be retested.

Furthermore, the program changes checkout function to call promotion, which changes the invariant in the postcondition of checkout from totalCost to a totalCost=orig(totalCost$*$(1-10%)). Every test that exercised checkout will need to be retested.

(4). If a function l in a library L in EC is changed, then all the functions in P that call l are affected, and all the tests exercising these functions need to be retested. Note that if the types of changes to L are unknown, we take a conservative approach that considers all the functions affected by these library calls, and selects all the tests exercising these functions to retest. Figure 4 shows an example of the library change, where the function isAfterPayDay takes datetime object as an input, which is an object from the Joda time library and calls its method getDayOfMonth(). When this library function is changed or its postcondition is changed, the precondition of isAfterPayDay is affected and the tests executing this function need to be retested. Table 4 shows the invariants in the pre- and postconditions of this library call.

```
public boolean isAfterPayDay(DateTime
datetime) {
  if (datetime.getMonthOfYear() == 2) {
// February is month 2!!
    return datetime.getDayOfMonth() > 26;
  }
  return datetime.getDayOfMonth() > 28;
}
```

Figure 4. An example of library change.

TABLE IV
THE INVARIANTS AFTER THE LIBRARY CHANGE.

Program Point	Invariant
...	...
isAfterPayDay.Enter	Datetime.getClass(). getName() = org.joda.time.DateTime
isAfterPayDay.Exit	Return in {true,false}
org.joda.time.DateTime. getDayOfMonth(): Enter	todayOfMonth is not null
org.joda.time.DateTime. getDayOfMonth(): Exit	dayOfMonth =orig(dayOfMonth)

(5). If a database schema is changed, then all the entity objects associated with the changed table are affected, and all the tests exercising these objects need to be retested. Most modern software applications use Object Relational Mapping to create relationships between database entities and program entities. In the example shown in Figure 5, the class Product maps to Table $T\_PRODUCT$ and its attributes id and price map to Table $T\_PRODUCT$'s column id and price, respectively. The hibernate uses a configuration file to create this mapping; when the database schema changes, we can identify the affected program entities from the mapping in the configuration file. In this example, when the price column in database $PRODUCT$ table is changed, then the attribute, price of Product, is affected. Every test that has price in the precondition of the function calls will need to be retested.

```
@Entity
@Table(name = "T_PRODUCT")
public class Product{
    @Column(name = "ID")
    private int id;
@Column(name = "PRICE")
    private float price;
    .....
}
```

Java Spring framework

```
<hibernate-mapping>
 <class name="Product" table="T_PRODUCT">
     <id name="id" column="ID">
        <generator class="increment" />
     </id>
   <property name="price" column="price" />
```

```
    ...
</class>
</hibernate-mapping>
```

Hibernate configuration file

```
public class ShoppingCart{
   Private float totalCost;
   Private List<Product> products;
    ....other attributes and functions
    Public void removeProduct(Product p ){
    if(p in products){
        products.remove(p)
        totalCost-=p.price
    }
}
    Public void checkout(){
    return totalCost;
    }
}
```

Figure 5. An example of database change.

TABLE V
THE INVARIANTS AFTER THE DATABASE CHANGE.

Program Point	Invariant
...	...
ShoppingCart.remove Product:Enter	p $\neq$ null
ShoppingCart.remove Product:Enter	p.price $\geq$ 0
ShoppingCart.remove Product:Exit	totalCost $\geq$ 0
ShoppingCart.remove Product:Exit	Products.size $\leq$ orig(Products. size)

Given a program P and a regression test suite T, we first run Daikon with P on T to obtain the likely invariants. Daikon infers invariants at the program points including function entries and exits. We run Daikon with P on T to obtain a set of invariants $I = < p, v, c >$, where p is the program point, v is the variables involved, and c is the condition. To keep track of the relationship between the invariants and the test cases, we create an invariant traceability matrix (ITM) by extending the invariant I to depict which invariants appear in each test case. A row in ITM shows the occurrence of an invariant in each test in T, and a column shows which invariants appear in a test case. This process can be done during the testing phase, and ITM can be created and updated after each test execution, or after testing phase and before any maintenance activity. When a change is made to P, a new test set T' is created to test the modified program P', Daikon is executed with P' on T' to obtain a set of invariants I'. Next, by comparing I and I, we can identify a set of affected invariants I_{af}.

- *If $i \in I$ and $i \notin I'$ then $i \in I_{af}$.*
- *If $i \in I'$ and $i \notin I$ then $i \in I_{af}$.*
- *If $i \in I'$ and I , $i.p = i'.p$, $andi.v = i'.v$, but i.c is not i'.c, then $i \in I_{af}$.*

If the change is in the execution context EC (including libraries, APIs, configuration files, and databases), or if an invariant involves a library function or API call, an entity object associated with a changed database entity, or a variable defined in a changed configuration file, the invariant is affected. The rules for the regression test selection are described above, and the algorithm is given in Figure 6.

```
Result: $T'$: a set of selected test cases
$T$ = $t_0$, $t_1$, .., $t_n$: a test suite of $P$;
$ITM[][]$: Invariant Traceability Matrix of $P$ w.r.t. $T$;
$I_{af}$: a set of affected invariants ;
$L$: a set of modified library functions;
$D$: a set of modified database tables;
$T'$ = ;
$i = 0$;
while $i < |T|$ do
    for each $inv \in I_{af}$ and $t_i \in T$ do
        if $ITM[inv.index][i] == 1$ then
            $T' = T' \cup \{ti\}$;
        end
    end
    for each $d \in D$; $obj$ is associated with d, and $inv$ is
      associated with $obj$ do
        if $ITM[inv.index][i] == 1$ then
            $T' = T' \cup \{ti\}$;
        end
    end
    for each $l \in L$; $obj$ is associated with l, and $inv$ is
      associated with $obj$ do
        if $ITM[inv.index][i] == 1$ then
            $T' = T' \cup \{ti\}$;
        end
    end
    i++;
end
return $T'$
```

Algorithm 1: TestCaseSelection

Figure 6. The selective regression algorithm.

A. Complexity Analysis

1. The execution time required for running Daikon and building ITM is $O(|V|^3 * |P| * |T|)$, where $|V|$ is the number of variables at the program points, $|P|$ is the number of the instrumented program points, and $|T|$ is the number of test cases [14].
2. The execution time required for comparing I and I' and identifying affected invariants is $O(|V|^3 * |P|)$
3. The execution time required for the test selection is $O((|P| * |V|^3)^2)$, where the algorithm parses ITM and I_{af} requiring $O(|I_{af}| * |ITM|)$ and is $O((|P| * |V|^3)2)$
The execution required by CART is $O(|V|^3 * |P| * |S|) + O(|P| * |T| * |V|^3) + O(|V|^3 * |P| * |S|') + O(|P| * |V|^3) + O((|P| * |V|^3)^2)$.

III. EMPIRICAL STUDIES

We conducted four empirical studies to evaluate the efficacy of our approach. CART targets different types of changes as

opposed to the existing approaches that account for single type of change only. The research questions we asked in these studies are: (1) for code changes, is CART as effective as the traditional approaches? (2) Is CART a safe approach, i.e., can it select all the fault-revealing regression tests? (3) For changes in the execution context, can CART select all the affected test cases, i.e., when the assumption of the constant factors external to the program is not held, is CART safe? (4) Can CART account for various types of changes at the same time? To answer the first question, we compared CART with the prevailing approaches that identify modification-traversing test cases by comparing the source code before and after a change. For the second question, we conducted experiments on three open source programs that have reported regression faults, and used these regression faults to evaluate if the tests selected by CART can detect these regression faults. To answer the third question, we upgraded and downgraded libraries that an application depends on and checked if CART selected all the test cases affected by the change of the library and potentially have backward/forward compatibility issues. Furthermore, we modified the database of an application and investigated if CART identified all the test cases that referenced the modified database entities. To address the last question, we conducted an experiment on software that has gone through several changes in code and execution context to demonstrate how CART takes care of multiple types of changes.
We developed a prototype of CART and used it to conduct the empirical studies on five open source software: Commons Lang, Commons IO, Commons Validator, and JXPathobtained from Apache Software Foundation, and BookStore from Github. Apache Software Foundation provides the release history and a source repository in which we can get source code of each release and the test suite for each project. Between two versions of the program there are some modifications for bug fixing or for improving the software performance. For each software, we selected four versions. Among GitHubs online project hosting services, we selected BookStore, which provides issue tracking and test cases.

These studies were conducted in two phases. The first phase is to observe the relationships between the likely invariants and the test cases. Daikon was used to instrument the subject software and to run the regression test suite to construct the invariant traceability matrix (ITM).

A. *Empirical Study I*

The first study was to demonstrate our selective regression testing after the code changes in the software. In this study, Commons Lang, Commons IO, Commons Validator from Apache Software Foundation were used. Table 6 shows the details of these systems. Commons Lang is a complement to standard Java libraries. It provides additional functions for manipulating the core classes of standard Java libraries. We used version 3.3.0 as the base version, then compared it with versions 3.3.1, 3.3.2, and 3.3.3, respectively. Commons IO is a library that provides useful functions that help to develop IO operations. The version 2.0 was used as the base version, and compared with versions 2.1, 2.2, and 2.3, respectively. Commons Validator is a package addressing the issues in data validation. This package helps to speed up the development and maintenance of validation rules. We used version 1.3.0 as the base version, then compared it with versions 1.4.0, 1.5.0, 1.6.0, respectively. Three approaches were used to evaluate the effectiveness: the re-test all was used to identify regression faults, CART and an execution trace based approach. We compared CART with an Execution Trace approach that identifies modification-traversing test cases by comparing the execution traces of the program before and after a change. A test was selected if it executed the statements that were changed in the modified program.

We first ran Daikon on the base version of the software with the corresponding test suite to obtain an array of invariants I and to create the invariant traceability matrix ITM. Then we performed the following steps for each modified version:

Step 1: First, we got the test suite T' that was created for testing the modified version. Each version of the software is associated with a test suite that was accumulated and developed, i.e., the test suite of the modified version contains tests used in the previous version. We filtered out these tests by comparing the two test suites and removing the ones in the test suite of the base version. Then we ran Daikon on the modified version with T' and obtained an array of invariants I'. Step 2: We compared$I$ andI' to obtain an array of affected invariants I_{af}: If then i I_{af}. Step 3: We ran the modified version of the software with the base test suite T and recorded the failure test cases caused by the regression faults. This step uses the retest-all approach to identify all the regression faults in order to determine if the approaches are safe. Step 4: We used the TestCaseSelection algorithm shown in Figure 1 to select test cases from T and retested the modified version with the selected test cases. We recorded the failure test cases that were caused by the regression faults. Step 5: We used the execution trace approach ET to select test cases and retested the modified version and recorded the failure test cases. Table 7 shows the results of Commons Lang, where all the regression faults detected by the retest-all method were detected by both CART and ET methods. The results are shown in Tables 6, 7, 8, and 9. In this study, both CART and ET were able to select all the fault-revealing test cases: all the regression faults detected by the retest were detected. Furthermore, CART selected fewer test cases than ET in all cases. In the Lang project, the number of test cases selected by our method was less than that of execution trace. In version 3.3.1, CART selected 379 test cases (19.87%) and ET selected 527 test cases (27.71%). The changes in this version were minor, such as the use of Long.decode(str) to replace Long.valueOf(str) and the removal of brackets to make the program clearer. For example, a change from $((u == 0)||(v == 0))$ to $(u == 0||v == 0)$. These changes do not affect the program behavior, and CART ignored such changes and selected only 20% of the test cases. In version 3.3.2 and version 3.3.3, the number of changed functions was four times greater than those of version 3.3.1. In version 3.3.2 almost all the scope of the functions parameters

TABLE VI
THE DETAILS OF THE SUBJECT PROGRAMS.

Program	Lang				IO				Validator			
Version	3.3.0	3.3.1	3.3.2	3.3.3	2.0	2.1	2.2	2.3	1.3.0	1.4.0	1.5.0	1.6.0
Class	86	99	107	132	100	102	102	103	38	57	62	63
Function	4306	4344	4625	4725	1874	1898	2010	2049	875	953	1186	1222
LOC	50766	58967	54064	55064	22042	22231	23788	23982	8632	10033	15354	16492
number of Test cases	1902	2051	2392	2505	744	789	931	952	524	415	613	657

and variables were changed to final. As a result, many test cases were selected by ET (64.41% - 73.55%). For program invariants, changes in scope will affect program invariants and the relationship between invariants; thus CART also selected many test cases (63.4% - 64.9%). The results are shown in Table 7.

In the IO project, the number of test cases selected by CART was between 16% and 37%, while ET selected 20% to 48.9% of test cases. As a result of new features being added to the newer programs, modifications were made to allow the program to utilize these features. The program invariants in the postcondition were affected. There are some changes that do not affect program behavior, such as a change in the name of a variable in an inner function, which would not affect its container class. The results are shown in Table 8.

In the validator project, CART selected around 20% of the test cases and ET selected around 42% of the test cases. We compared the differences between the different versions of the program. Among the hundred lines of the changed code, many changes did not affect the program behavior. For example, in the class CreditCardValidator, four functions were changed, Visa, Amex, Discover, and Mastercard, from private to private static, which improved the program performance by saving memory space for creating instances of these four functions, but did not change the program behavior. Another example is, in the class UrlValidator, the library Perl5Util was used to validate the URL. However, it was replaced by regex.Matcher in the newer version, which also did not affect the program behavior. Thus CART selected much fewer test cases than ET. The results are shown in Table 9.

In summary , CART selected fewer test cases than did ET, and still selected all the fault-revealing test cases. When modifications are made to the program and do not change the program behavior, such as cleaning the codes, changing a variables name in a function or changing the implementation of some functions to improve the performance, e.g. changing a variables type from String to String builder or adding static scope to save the memory, these changes do not affect program behavior, and thus, were not be selected by CART.

B. Empirical Study II

The second study focused on the changes in the programs dependent libraries. When a library is changed, in most cases, it is difficult to identify where the changes are and how the program will be affected. Our approach selects all the tests that use the changed library. In our experiment, Daikon generated program invariants based on every statement executed, including calls to library functions, informing us of library calls used in each test case. We used Lang, Validation, and JXPath from Apache Software Foundation. All three programs are maven projects, and the dependent libraries can be easily changed by modifying the pom file. In the Lang and JXPath projects we downgraded and upgraded one of their dependent libraries. For Validation we downgraded and upgraded two versions of its dependent libraries.

The regression test selection was conducted as follows:

Step1: We ran Daikon on the program with the base version of the library and the test suite T to obtain an array of invariants I and to create ITM.

Step2: We used the changed library as input in TestCaseSelection algorithm to select test cases from T and to get T'.

Step3: We ran the program with the changed dependent library and tested suite T to record the affected test cases in T'' (failed in the changed library).

Step 4: We compared T' and T''.

The results of regression testing on this study are summarized in Tables 10, 11, and 12. All the experiments show that all the affected test cases (cases that failed in the changed library) were selected by our approach; thus, it is safe with respect to the library changes. We observed that many libraries are backward but not forward compatible. In the programs Lang and Validation, after we upgraded the dependent libraries, easy mock and digester, the changes did not affect the programs behavior. When the dependent libraries were downgraded, in the Lang project, there were three affected test cases, which were assertion failures caused by the changes inside the library. In Validation and JXpath, we observed that the upgraded and downgraded libraries removed or changed some functions, which caused many errors in the calling functions in the program. Thus, many test cases failed.

TABLE X
THE RESULTS OF THE CHANGED LIBRARY EASYMOCK.

Program	Lang 3.3.0	
Changed Library	easymock	easymock
Modification	2.5.2 -3.4	2.5.2 -2.0
Type	Upgrade	Downgrade
Regression faults	0	3
Selected Test Cases	9	9
Detected regression faults	0	3

TABLE VII
THE RESULTS OF THE LANG PROJECT.

Program	Lang-3.3.0					
Version	3.3.1		3.3.2		3.3.3	
Modified classes	23		36		39	
Modified functions	31		139		290	
Modified statements	236		2073		2234	
Regression faults	4		7		24	
Selected test cases	CART	ET	CART	ET	CART	ET
	379 (19.87%)	527 (27.71%)	1235 (64.93%)	1399 (73.55%)	1206 (63.41%)	1225 (64.41%)
Detected regression faults	4	4	7	7	24	24

TABLE VIII
THE RESULTS OF THE IO PROJECT.

Program	IO 2.0					
Version	2.1		2.2		2.3	
Modified classes	7		7		12	
Modified functions	18		18		37	
Modified statements	93		93		297	
Regression faults	4		4		6	
Selected test cases	CART	ET	CART	ET	CART	ET
	121 (16.26%)	150 (20.16%)	127 (17.07%)	150 (20.16%)	273 (36.69%)	364 (48.92%)
Detected regression faults	4	4	4	4	6	6

TABLE XI
THE RESULTS OF THE CHANGED LIBRARIES BEANUTIL AND DIGEST.

Program	Validation			
Changed Library	beanutils		digester	
Modification	1.8-1.9.3	1.8-1.6.3	1.8-2.1	1.8 -1.3
Regression faults	121	121	0	90
Selected Test Cases	121	121	90	90
Detected regression faults	121	121	0	90

TABLE XII
THE RESULTS OF THE CHANGED LIBRARY JDOM.

Program	JXPath	
Changed Library	jdom	jdom
Modification	1.1-2.02	1.1-b7
Regression faults	30	23
Selected Test Cases	30	30
Detected regression faults	30	23

C. *Empirical Study III*

The third study focused on the database change. Many applications use databases to store persistent data. Any change of the database structure will affect program behavior. Modern software applications use object-relational mapping (ORM) to map database tables to entity objects in a program. When a database structure is changed, we can use ORM to identify which entity objects are affected, and select test cases referring to the changed entity objects to re-test. In this study, we used a web application BookStore from GitHub. BookStore is an online shopping platform for searching books, checking out, etc. This web application stores data in MySQL database, uses com.mchange.v2.c3p0 package and javax.sql to transform data from MySQL database into entity objects. We made three modifications to the database. The first one changed the name column of the book table into title. Upon initializing the book entity object, this change resulted in the program failing to find the attribute name. Because the book table is mapped to the book entity object, we could locate the book entity in program invariants. Any test case referring to the book entity will be selected. The second change deleted the category table, which is used for categorizing books. After deleting this table, all the books were categorized into one group. Because the category table is associated with the category entity object, all the test cases referring to this object were selected. The third modification added a column into the book table. Since the new column has not been used, it does not affect the program. The results of these three database modifications are shown in Table 13. For the first modification, our method selected 31 test cases; by retesting all the test cases, three test cases failed and were selected by our method. The recall is 1 and the precision is 0.01. For the second modification, our method selected 62 test cases, by retesting all the test cases, 12 of which failed and were selected by our method. The recall is 1 while the precision is 0.19. For the last one, our method selected 31 test cases, by retesting all the test cases, which resulted in zero failures. In summary, our approach successfully selected test cases affected by the database change. The precision is low, due to the use of mock data in the test case rather than real data in the database.

D. *Empirical Study IV*

The aim of this study was to investigate the effectiveness of CART when there are heterogeneous types of changes taking place at the same time. We used an educational software that facilitates students, teachers, and researchers to acquire and share information across knowledge building communities. This system has been used in many countries for seven years and has continuously evolved to provide better services. ITM was designed using a multi-tiered web architecture and implemented in D3.js, JavaScript, JSP, Java, and MySQL. The

TABLE XIII
THE RESULTS OF THE DATABASE CHANGES.

Database Change	Modify column	Delete table	Add column
Regression faults	3	12	0
Selected tests	31	62	31
Detected regression faults	3	12	0

version used in this study contains 39 classes, 494 functions, and 4,668 LOC, and there are 96 test cases in the regression test suite. There are six modifications in this version including:

- Change in Search function that allows search content by view and project together.
- Change getLocalCommunity in CommunityService to add token info in community.
- Change in Database schema that changes user tables primary Id from int to String.
- Change operation, getProjectByNotes to add NULL Point Check for GetThreadByNote.
- Change operation, getProjectsByUserId that changes the SQL used to query the project.
- 6.Change Library from fastjson-1.1.41.jar to json-simple-1.1.1.jar.

We first ran the 96 test cases on the modified software and detected two regression faults, then we used CART to select 47 test cases and detected the two regression faults as well. Next, we used ET to select 40 test cases, but could not detect any regression fault. In this study we observed that (1) CART selected more test cases than ET, and found that the same invariant involved a class that had different instances. When the invariant with one of the instance was affected, all the tests that included this invariant regardless of which instance it involved was selected, causing the selection of some tests that were not affected by the changes. (2) The ET approach failed to detect the two regression faults, making it unsafe when there are changes in the execution context.

E. Empirical Study IV

To automatically obtain pre- and postconditions of the functions, our approach relies on Daikon, which has several limitations. First, like all machine learning techniques, Daikon requires sufficient data to infer likely invariants. If the regression test suite is too small, then the obtained invariants may be incomplete or inaccurate. However, if the regression test suite is small, then there is no need for selection. Retesting all test cases should not be too costly. Second, Daikon may produce too many invariants that are irrelevant to the program behaviors. Using all the invariants produced by Daikon may result in selecting too many regression tests. Thus, some filtering of the spurious invariants will be needed. Third, some languages, such as JavaScript, that are commonly used for the frontend development and are strongly dependent on the runtime libraries are not supported by Daikon. At this point, the invariants from JavaScript cannot be obtained automatically and need to have manually defined pre- and postconditions for the functions. Fourth, the same invariant may be inferred by the different instances of the objects, which may cause CART to select test cases that are not modification-traversing.

IV. RELATED WORK

Techniques for regression testing have been well studied. Existing approaches can be classified into three areas: minimization, prioritization, and selection. The minimization [4], [13], [18], [19] and the prioritization approaches [9], [29], [33] aim at selecting a minimum or high priority subset from the test suite to cover the critical parts of the program that need to be retested. They first identify portions of a program that should be retested, and then select some tests that will exercise these parts and satisfy certain criteria. The minimization and the prioritization methods can reduce the number of test cases to be retested, but they may overlook some fault-revealing test cases [27]. The selection approaches [34], [1], [2], [4], [37], [27] on the other hand will select all the fault-revealing test cases, thus guaranteeing the quality of the software. However, it requires a longer processing time and a larger size of the retest test suite in general. In this section, we provide an overview for the existing selection approaches. The slicing method is widely used in regression testing. Execution, dynamic and relevance slicing [1], [3] are typical slicing methods, among which execution and relevance slicing are safe while dynamic slicing is not. Execution slicing regression testing will only retest the test cases which execute the modified codes. It has been adopted in many regression testing tools due to its simplicity and efficiency. The execution slicing technique has many variations; it can be based on statement, block or function level. Regardless of which variations, precision is a major problem that is difficult to conquer. To solve this problem, Wong et al. [38] proposed a methodology that takes the coverage information into account, or assigns priorities to different test cases, so that a smaller subset of the test cases, based on these criteria, can be selected. Dynamic and relevance slicing only retest the test cases in which the behavior of the system is really affected by the modification. Dynamic slicing will only consider the non-predicate statements which affect the output, whereas relevance slicing considers all the predicate statements which will affect the output. Dynamic slicing is more precise than the others, but in most cases, it is not as safe.

Rothermel and Harrold [28], and Ball [2] provide a regression approach for both procedure programs. They use the control flow graph(CFG) to represent a program. After a modification, the CFGs of the program before and after the modification are compared. It traverses the two CFGs and compares pairs of nodes in the two graphs. Starting with the two root nodes, it compares the successors of the pair of the nodes with identical labeled edges. If the two successor nodes are not equivalent (have different labels), then the tests that exercise this edge are considered modification-traversing. They propose a group of methods which can be classified into basic algorithms and algorithms with added precision. The basic algorithm is similar to execution slicing and the algorithms with added precision also consider the effects of the modifications on the output of the system.

The firewall approaches [17], [37], [34], [36], [35], which identify the set of modules that may be affected by the modification, then perform unit testing for the modified modules and integration testing on the set of modules which have been selected. Chen et al.[6] developed a tool, TestTube, which will identify modified/affected entities (functions, types, variables and macros). Then it retests all the test cases which include at least one affected entity.

Other approaches, such as the model-based approaches [15], [16], use UML design models to determine the affected use cases, classes, states, and diagrams etc. based on their relationship with the changed design artifacts. Orso et al. [23] proposed a partition based approach for java programs. Their approach constructs Interclass Relation Graphs (IRG) for the program before and after a modification. The IRG is similar to a class diagram which contains hierarchical, aggregation and the relation between classes and interfaces. The dataflow based approaches [20, 47] use dataflow information to augment Control Flow Graph by adding global variables, function parameters and a return value of functions. Configuration based approach [39], [40], [41] aims to select test cases to execute the code in the new configuration that is not tested in the old one. A configurable system uses different parts of the system based on a predefined set of the configurations. When a configuration is changed, it compares the differences between the two configurations and uses program slicing to determine which functions are affected (not covered by the old one) by the change of the configuration. The test cases that exercise an affected function are selected for regression testing.

For non-code based changes, Haraty et al. [12] presented a two-phase methodology for regression testing SQL-based database applications. They first built control flow graphs for database modules consisting of SQL compound statements, and performed dataflow analysis on database columns. Based on this analysis, Phase 1 identified modifications and performed change impact analysis. Phase 2 used two algorithms for regression test selection: the Graph Walk algorithm traverses the control flow graph of database modules and selects a safe set of test cases to retest. The Call Graph Firewall algorithm uses dataflow analysis to build a firewall at the inter-procedural level, which includes the modified database components in the firewall. Nanda et al. [22] presented an approach to address regression test selection when there is no code change, but property files or databases are changed. They first built abstract models for configuration files and databases and compared the models before and after the changes to identify modified entities. Then, they monitored each execution to build a traceability that links the test cases to the accessed property and the issued database commands. Based on the modified entities and the traceability, they selected tests that traverse the modified entities to retest.

V. CONCLUSIONS AND FUTURE WORK

We have presented a new selective regression technique that accounts for changes in programs as well as in the execution context. Our technique is inclusive; not only is it safe in controlled regression testing, but also selects all fault-revealing tests when the external factors are not constant. It is efficient, because it only maintains the invariant traceability matrix, which is bounded by the number of functions multiplied by the number of test cases. For precision, it may select more test cases than the execution trace based techniques that select modification-traversing test cases when the modified statements are in one of the many branches in a function. On the other hand, it may be more precise than the execution trace based approaches when the modification does not impact the behavior of the function, and all of the invariants hold after the modification.

To improve the applicability of CART, we are working on defining invariants for other languages that are not currently supported by Daikon, such as JavaScript, that are frequently used for implementing frontends and heavily rely on the runtime library. To improve the precision, we are studying what types of invariants are more sensitive to changed behavior and aim to only use these invariants to improve the precision of CART. In the meantime, we are modifying Daikon to address the polymorphism issue of the invariants.

REFERENCES

[1] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. A. London. Incremental regression testing. In *Software Maintenance, 1993. CSM-93, Proceedings., Conference on*, pages 348–357. IEEE, 1993.

[2] T. Ball. On the limit of control flow analysis for regression test selection. *ACM SIGSOFT Software Engineering Notes*, 23(2):134–142, 1998.

[3] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 384–396. ACM, 1993.

[4] T. Y. Chen and M. F. Lau. Dividing strategies for the optimization of a test suite. *Information Processing Letters*, 60(3):135–141, 1996.

[5] Y. Chen, M. Ying, D. Liu, A. Alim, F. Chen, and M.-H. Chen. Effective online software anomaly detection. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 136–146. ACM, 2017.

[6] Y.-F. Chen, D. S. Rosenblum, and K.-P. Vo. Testtube: A system for selective regression testing. In *Proceedings of the 16th international conference on Software engineering*, pages 211–220. IEEE Computer Society Press, 1994.

[7] J. Cobb, J. A. Jones, G. M. Kapfhammer, and M. J. Harrold. Dynamic invariant detection for relational databases. In *Proceedings of the Ninth International Workshop on Dynamic Analysis*, pages 12–17. ACM, 2011.

[8] Z. Ding, M.-H. Chen, and X. Li. Online reliability computing of composite services based on program invariants. *Information Sciences*, 264:340–348, 2014.

[9] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE transactions on software engineering*, 28(2):159–182, 2002.

[10] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.

[11] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th international conference on Software engineering*, pages 291–301. ACM, 2002.

[12] R. A. Haraty, N. Mansour, and B. Daou. Regression test selection for database applications., 2004.

[13] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2(3):270–285, 1993.

[14] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for java software. In *ACM Sigplan Notices*, volume 36, pages 312–326. ACM, 2001.

[15] M. Z. Z. Iqbal, Z. I. Malik, A. Nadeem, et al. An approach for selective state machine based regression testing. In *Proceedings of the 3rd international workshop on Advances in model-based testing*, pages 44–52. ACM, 2007.

[16] M. Z. Z. Iqbal, Z. I. Malik, M. Riebisch, et al. A model-based regression testing approach for evolving software systems with flexible tool support. In *Engineering of Computer Based Systems (ECBS), 2010 17th IEEE International Conference and Workshops on*, pages 41–49. IEEE, 2010.

[17] D. C. Kung, J. Gao, P. Hsia, J. Lin, and Y. Toyoshima. Class firewall, test order, and regression testing of object-oriented programs. *JOOP*, 8(2):51–65, 1995.

[18] J.-W. Lin, C.-Y. Huang, and C.-T. Lin. Test suite reduction analysis with enhanced tie-breaking techniques. In *Management of Innovation and Technology, 2008. ICMIT 2008. 4th IEEE International Conference on*, pages 1228–1233. IEEE, 2008.

[19] N. Mansour and K. El-Fakih. Simulated annealing and genetic algorithms for optimal regression testing. *Journal of Software: Evolution and Process*, 11(1):19–34, 1999.

[20] L. Mariani and M. Pezze. A technique for verifying component-based software. *Electronic Notes in Theoretical Computer Science*, 116:17–30, 2005.

[21] S. Mostafa, R. Rodriguez, and X. Wang. Experience paper: a study on behavioral backward incompatibilities of java software libraries. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 215–225. ACM, 2017.

[22] A. Nanda, S. Mani, S. Sinha, M. J. Harrold, and A. Orso. Regression testing in the presence of non-code changes. In *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, pages 21–30. IEEE, 2011.

[23] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 241–251. ACM, 2004.

[24] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, et al. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 87–102. ACM, 2009.

[25] R. Pietrantuono, S. Russo, and K. S. Trivedi. Online monitoring of software system reliability. In *Dependable Computing Conference (EDCC), 2010 European*, pages 209–218. IEEE, 2010.

[26] N. Polikarpova, I. Ciupa, and B. Meyer. A comparative study of programmer-written and automatically inferred contracts. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 93–104. ACM, 2009.

[27] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(2):173–210, 1997.

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

[29] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on software engineering*, 27(10):929–948, 2001.

[30] S. K. Sahoo, J. Criswell, C. Geigle, and V. Adve. Using likely invariants for automated software fault localization. *ACM SIGPLAN Notices*, 48(4):139–152, 2013.

[31] D. Schuler, V. Dallmeier, and A. Zeller. Efficient mutation testing by checking invariant violations. In *ISSTA '09: Proceedings of the 18th International Symposium on Software Testing and Analysis*, pages 69–80, July 2009.

[32] F. I. Vokolos and P. G. Frankl. Empirical evaluation of the textual differencing regression testing technique. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 44–53. IEEE, 1998.

[33] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos. Timeaware test suite prioritization. In *Proceedings of the 2006 international symposium on Software testing and analysis*, pages 1–12. ACM, 2006.

[34] L. White and K. Abdullah. A firewall approach for regression testing of object-oriented software. *Software Quality Week*, 27, 1997.

[35] L. White, H. Almezen, and S. Sastry. Firewall regression testing of gui sequences and their interactions. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 398–409. IEEE, 2003.

[36] L. White, K. Jaber, B. Robinson, and V. Rajlich. Extended firewall for regression testing: an experience report. *Journal of Software: Evolution and Process*, 20(6):419–433, 2008.

[37] L. J. White and H. K. Leung. A firewall concept for both control-flow and data-flow in regression integration testing. In *Software Maintenance, 1992. Proceerdings., Conference on*, pages 262–271. IEEE, 1992.

[38] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *Software Reliability Engineering, 1997. Proceedings., The Eighth International Symposium on*, pages 264–274. IEEE, 1997.

[39] J. Zheng, B. Robinson, L. Williams, and K. Smiley. An initial study of a lightweight process for change identification and regression test selection when source code is not available. In *Software Reliability Engineering, 2005. ISSRE 2005. 16th IEEE International Symposium on*, pages 10–pp. IEEE, 2005.

[40] J. Zheng, B. Robinson, L. Williams, and K. Smiley. Applying regression test selection for cots-based applications. In *Proceedings of the 28th international conference on Software engineering*, pages 512–522. ACM, 2006.

[41] J. Zheng, B. Robinson, L. Williams, and K. Smiley. A lightweight process for change identification and regression test selection in using cots components. In *Commercial-off-the-Shelf (COTS)-Based Software Systems, 2006. Fifth International Conference on*, pages 7–pp. IEEE, 2006.