Regression Testing in the Presence of Non-code Changes

Agastya Nanda*, Senthil Mani†, Saurabh Sinha‡, Mary Jean Harrold‡, and Alessandro Orso‡
{*Jaypee Institute of Information Technology
Email: agastya.n@gmail.com
†IBM Research – India
Email: {sentmani|saurabhsinha}@in.ibm.com
‡Georgia Institute of Technology
Email: {harrold|orso}@cc.gatech.edu

Abstract—Regression testing is an important activity performed to validate modified software, and one of its key tasks is regression test selection (RTS)—selecting a subset of existing test cases to run on the modified software. Most existing RTS techniques focus on changes made to code components and completely ignore non-code elements, such as configuration files and databases, that can also change and affect the system behavior. To address this issue, we present a new RTS technique that performs accurate test selection in the presence of changes to non-code components. Our technique computes traceability between test cases and the external data accessed by an application, and uses this information to perform RTS in the presence of changes to non-code elements. We present our technique, a prototype implementation of the technique, and a set of preliminary empirical results that illustrate the feasibility, effectiveness, and potential usefulness of our approach.

I. INTRODUCTION

Regression testing is performed on modified software to provide confidence that the changes behave as expected and that the unchanged parts of the code are not adversely affected by the modifications. Reports estimate that regression testing consumes as much as 80% of the overall testing budget and can consume up to 50% of the cost of software maintenance [1]–[3]. One of the key tasks in regression testing is regression test selection (RTS): selecting a subset of the existing test cases to rerun on the modified software in order to reduce the costs of regression testing. In cases where a large number of test cases are unrelated to the changes, and therefore irrelevant for the detection of regression faults, RTS can provide significant cost savings over running all the existing test cases.

To date, researchers have presented many RTS techniques (e.g., [4]–[10]). Although these techniques differ from one another in various respects, such as cost, precision, and safety,1 they all share the common characteristic of being code-centric: they focus on changes made to the application code and completely ignore changes made to non-code components. Thus, these techniques work well for the kind of traditional software they target—homogeneous and monolithic software.

1An RTS technique is safe if it guarantees that all test cases that may reveal a different behavior in the modified program are selected [5].

Modern software systems, however, are often very different from these traditional systems: they consist of a number of components of diverse nature, with different degrees of coupling with one another, that are often distributed and highly dynamic. Most existing RTS techniques are not applicable to these kinds of systems. In fact, regression testing of these systems is often performed in ad-hoc, inadequate ways, which can have dramatic consequences. For example, Amazon’s day-long outage in 2008 was “due to an upgrade of the company’s Web site” and cost the company millions of dollars [11].

At an abstract level, the main differences between today’s systems and traditional software can be distilled into two characteristics: heterogeneity and environment dependence. Heterogeneity occurs because components may come from different sources, be written in different languages, and be available in different formats (e.g., as source code, as binary code, or through remote interfaces). Environment dependence exists because the behavior of these systems can be affected not only by changes in their code, but also (and sometimes to a larger extent) by changes in their complex environments (e.g., databases, configuration files, and network layouts).

In this paper, we concentrate on this latter characteristic and present a novel test-selection technique that aims to overcome the limitations of existing RTS techniques in handling software whose behavior is affected by non-code entities—in particular, by configuration files and databases. Numerous classes of applications that are increasingly popular today, such as web, enterprise, and server applications, have these characteristics. For these applications, purely code-centric approaches are inadequate, as they ignore changes to external entities, which can cause regression faults even in the absence of changes to the code.

Changes in configuration files are problematic because they can affect a software system’s behavior in many ways, ranging from the layout of a graphical interface to the set of protocols that a server supports. Database-related changes can also be problematic and affect the behavior of a system that interacts with the changed database in several ways. The database schema could change (e.g., a column may be added, deleted, or modified in a table), potentially affecting test
cases that access that schema. Or, a database trigger could be modified, which would affect the behavior of all those test cases whose database interactions cause the modified trigger to be executed. Effective test selection in the presence of these types of changes requires not only the detection of the changes, but also the creation of appropriate traceability between test cases and external entities.

Our approach addresses these issues by performing different types of analyses. First, it builds abstract models of configuration files and databases and performs differencing on these models to identify the modifications made to these external entities. The approach can perform differencing at different levels of granularity and is able to accommodate changes to different types of entities. For configuration files, for instance, the analysis could identify just the modified files or perform a finer-grained analysis that detects the particular values (or properties) that changed.

Second, our approach creates a new form of traceability that links test cases to external entities. Conventional traceability analysis associates test cases with code entities (e.g., methods and statements) only. To consider also external entities, our analysis monitors the execution of a program and keeps track of (1) the properties that are accessed and (2) the database commands that are issued. Using the captured properties and commands, our analysis associates each test case with the properties and database entities accessed during the execution of that test case. Similar to the differencing analysis, the traceability analysis can be performed at different levels of granularity, and must accommodate different types of entities (e.g., database tables and database triggers).

Finally, based on the modified entities and the traceability between test cases and entities, the approach selects the test cases that traverse the modified entities and that should, therefore, be rerun.

We implemented our technique in a prototype tool, TREDS. The current version of TREDS can perform RTS for Java applications that interact with text-based property files and MYSQL databases. To assess the feasibility, effectiveness, and potential usefulness of our approach, we conducted an empirical evaluation in which we used TREDS on two real systems: a proprietary application and an open-source system. Our results show that, for the two subjects considered, a code-centric technique can fail to select many test cases that are affected by changes to external entities and that are selected by TREDS. The results also show that simply extending a code-centric technique by selecting all test cases that traverse the changed entities and that would affect the behavior of all those test cases whose database interactions cause the modified trigger to be executed. Effective test selection in the presence of these types of changes requires not only the detection of the changes, but also the creation of appropriate traceability between test cases and external entities.

Figure 1 presents a general RTS system, which consists of three components: (1) a component for modeling and differencing that identifies the changed entities between two program versions, (2) a component for creating traceability between test cases and model entities, and (3) a component for selecting the test cases that cover the modified entities.

RTS in the presence of changes to non-code components of a system. Because the use of configuration files and databases is pervasive in modern software systems, our technique enables effective regression testing to be performed on a large class of modern software systems, such as web, enterprise, and server applications, for which existing RTS techniques are either ineffective or completely inapplicable.

The main contributions of this paper are:

- The description of a novel RTS technique that performs accurate test selection in the presence of changes to configuration files and databases.
- The description of an implementation of the technique in the TREDS prototype, which can handle Java applications that interact with text-based property files and relational databases.
- The results of an empirical study that illustrates the usefulness of the approach and shows that our technique can outperform existing RTS techniques and straightforward extensions of such techniques.

II. BACKGROUND AND MOTIVATING EXAMPLE

In this section, we present background material on regression test selection and illustrate the problem we target using an example application that accesses a configuration file.

A. Regression Test Selection

Given a program \( P \), a test suite \( T \) for \( P \), and a modified version \( P' \) of \( P \), the goal of regression test selection (RTS) is to select a subset \( T' \) of \( T \) to rerun on \( P' \). A safe RTS technique selects all test cases that can potentially reveal faults in \( P' \) (i.e., that are fault-revealing). Because finding fault-revealing test cases is undecidable, RTS techniques select modification-traversing test cases, which, under certain assumptions, include all fault-revealing test cases [5].
A straightforward extension to existing RTS techniques to handle such changes would mark as modified all the program points where a changed property file is loaded, and select the test cases that traverse these points. This approach would mark statement 9 of GetServerURL as modified. Then, DEJAVOO would identify edge (8, 9) as an affected edge and select the test cases associated with that edge, which includes all four test cases. This approach is imprecise because test cases $t_2$ and $t_4$ are selected unnecessarily. Therefore, to be accurate, RTS should select only those test cases during the execution of which the specific modified properties are read.

### III. Our Approach

RTS requires analyses to (1) identify the affected system entities between two system versions, (2) compute traceability between test cases and the system entities, and (3) select test cases that cover the affected entities. The code-centric techniques perform these analyses for code components only. Our approach overcomes this restriction to perform the three types of analyses for configuration files and databases.

Figure 3 presents our approach for regression test selection. In addition to the code-differencing component, the system includes components to compute differences between two versions $R$ and $R'$ of a property file and two versions $D$ and $D'$ of a database. These components build models of property files and databases, and compare two models to identify the affected model entities. The traceability-analysis component associates test cases with the model entities, where a model could represent code, property files, or databases. Finally, the test-selection component selects the test cases that cover the affected model entities.

#### A. Modeling and Differencing

1) **Modeling properties:** We model property files as a collection of properties, where each property is represented as a $(name, value)$ pair. The name of a property is qualified with the name of the file in which it occurs; this lets properties with the same name be specified in different files. For example, the properties for GetServerURL are represented as two pairs

```java
public class GetServerURL {
    public String getURL(String zone, String protocol)
        throws FileNotFoundException, IOException {
        String url = "*";
        if (zone.equals("INT") {
            propName = "URL_INTERNAL";
        } else {
            propName = "URL_EXTERNAL";
        }
        Properties prop = new Properties();
        prop.load(new FileInputStream("conf.property"));
        String value = prop.getProperty(propName);
        if (protocol != null) {
            url = protocol + value;
        }
        return url;
    }
}
```

![Figure 2. Example program GetServerURL: two versions of the property file read by the program, and a test suite for the program.](image)

Figure 2 shows two versions of the property file. In the second version, the value of `URL_INTERNAL` is modified. Although the program is unmodified, the change in the property file alters the behavior of the program: the program can construct a different URL than it did for the previous version of the property file. Thus, from the regression-testing perspective, test cases that are affected by the change should be rerun. Figure 2 also shows a test suite containing four test cases of which two test cases ($t_1$ and $t_3$) read the modified property. Because there is no change in the program, code-centric RTS techniques would not be effective in selecting these affected test cases. For example, because there are no affected edges, the DEJAVOO [5], [8], [9] system would not select the affected test cases.
The differencing analysis takes as input two property models $R$ and $R'$ and identifies the affected properties in $R$. A property $r \in R$ is affected if it does not occur, or occurs with a different value, in $R'$. In the former case, $r$ is deleted, whereas in the latter case, $r$ is modified.

2) **Modeling databases:** Modeling a database presents challenges because it requires diverse types of database entities to be considered. In addition to the data schema and the actual data content, databases consist of other elements such as triggers and stored procedures. Such elements can change and affect the behavior of an application that interacts with the database. Thus, they need to be modeled appropriately. Our current approach models database schema and triggers.

We model the database schema $D_s$ as a collection of tables or relations, where a table is defined as a list of columns, and each column has a type associated with it. One or more of the columns serve as the primary key of the table. More formally, we model a database as follows:

$$D_s = \{T_1, \ldots, T_n\}$$
$$T = \{(c_1, \tau_1, n_1), \ldots, (c_k, \tau_k, n_k)\}, \rho$$

In the model, $T$ is a table, $c$ is a column, $\tau$ is the type of a column, $n$ is a flag indicating whether the column can contain null values, and $\rho$ is the primary key of the table.

Given two versions $D_s$ and $D'_s$ of a database schema, our approach computes differences between the versions by analyzing changes in the elements of the schema. First, for each table $T$ in $D_s$, the differencing algorithm checks whether $T$ occurs in $D'_s$ by searching for a table with the same name. If $T$ does not occur in $D'_s$, it is marked as deleted. Otherwise, the algorithm checks whether a column of $T$ is deleted or modified. A column $(c, \tau, n)$ of $T$ is classified as modified if either $\tau$ or $n$ has changed. The algorithm also checks whether the primary key of $T$ has changed. If any of the elements has changed, $D_s$ is classified as modified. Depending on the element, the algorithm also records the old and new values. First, the algorithm compares $T$ and $T'$ and, if they are not equal, records the old and new tables to which the trigger is attached. Similarly, it compares $e$ and $e'$ and, if they are not equal, records the old and new events.

A database trigger is code that executes automatically in response to database events. Adding a record to a table, for instance, could cause a trigger to insert records in other related tables. Triggers can thus be defined to execute when the data in a table is modified. Triggers can also be defined at the schema-level; such triggers are executed when schema-level changes occur (e.g., the creation or deletion of a table). The syntax of triggers and the events in response to which triggers execute vary considerably between different database systems (e.g., DB2, Oracle, MS SQL server, and MySQL).

At an abstract level, we model a database trigger as a 4-tuple $D_t = (T, e, A, p)$, where $T$ is the table to which the trigger is attached, $e$ is the event (insert, update, or delete) on $T$ that causes the trigger to be executed, $A$ is a sequence of actions (i.e., the SQL commands) that execute in response to the event, and $p$ is a precedence specifying whether $A$ is performed before or after $e$.

### Trigger create_salary_rec

**version 1**

- $T = \text{Employee table}$
- $e = \text{INSERT record}$
- $A = \text{INSERT record into Salary table}$
- $p = \text{AFTER}$

**version 2 (A updated)**

- $A' = \text{INSERT record into Salary table; initialize values}$

**version 2 (e updated)**

- $e' = \text{UPDATE column is_confirmed = true}$

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Accesses Emp Table</th>
<th>Affected in Version 2</th>
<th>Affected in Version 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1 insert employee record</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>t2 print paystub</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>t3 update employee is_confirmed to true</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>t4 print employment record</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

Figure 4. Example to illustrate the effects of trigger modifications on test selection.

Figure 4 presents an example to illustrate the implications of trigger changes for regression test selection. The figure shows a trigger create_salary_rec, along with its four elements (version 1). This trigger gets executed after a new employee record is added to the Employee table; the trigger action inserts a record into the Salary table. (We omit the schema definition as it is not relevant for the discussion.)

Given two versions $D_t = (T, e, A, p)$ and $D'_t = (T', e', A', p')$ of a database trigger, the trigger-differencing algorithm compares each of the four elements comprising the triggers to identify the changed elements. If any of the elements has changed, $D_t$ is classified as modified. Depending on the element, the algorithm also records the old and new values. First, the algorithm compares $T$ and $T'$ and, if they are not equal, records the old and new tables to which the trigger is attached. Similarly, it compares $e$ and $e'$ and, if they are not equal, records the old and new events. Then, it compares $A$ with $A'$ and $p$ with $p'$ and determines whether they are changed. Each comparison is done as a simple string compare. The algorithm marks $D_t$ as affected if $D_t$ is either deleted or modified.

Figure 4 also shows two modified versions of the trigger. In version 2, the action of the trigger is modified to initialize the values of the inserted salary record. To illustrate the effects of this change on test selection, consider the test suite, consisting of four test cases, shown in the figure. Each test case accesses the Employee table. However, the change in version 2 affects only $t_1$, which inserts a record into the Employee table. The remaining test cases are unaffected and, therefore, need not be rerun.

In version 3 of the trigger, the trigger event is changed from INSERT to UPDATE of the Employee table, specifically the update of column is_confirmed of the table to true. This change affects the behavior of $t_1$ because the trigger is not executed after the insertion of the employee record (as it did previously). Additionally, the change affects the behavior of $t_3$, which updates the employee status column is_confirmed.
When executed with version 1 of the trigger, \( t_3 \) does not cause the trigger to be fired, whereas when executed with version 3, the update performed by \( t_3 \) causes the trigger to be executed. Therefore, for version 3, test cases \( t_1 \) and \( t_3 \) should be selected. Note that, in this case, test cases that satisfy the conditions of either the old event \( e \) or the new event \( e' \) need to be rerun.

### B. Traceability Analysis

1) **Traceability for properties:** Traceability analysis for properties requires test cases to be linked to the property files and/or properties that are accessed during test execution. We represent the traceability as \( [t, \{r\}] \), where \( t \) is a test case and \( r \) is the set of properties accessed during the execution.

Traceability can be computed using code instrumentation, based on the specific API used for interactions with properties. For the Java properties API \((java.util.Properties)\), the instrumentation records the file loaded at the load command (line 9 of the example in Figure 2) and the specific property accessed using the “get” command (line 10 in Figure 2). Thus, for that example, the traceability analysis associates test cases \( t_1 \) and \( t_3 \) with URL_INTERNAL, test cases \( t_2 \) and \( t_4 \) with URL_EXTERNAL, and all four tests with the property file conf.property.

2) **Traceability for database entities:** To discuss the traceability and test selection for databases, we first introduce some terminology. We use \( s \) to refer to an SQL query or update command: \( \text{tab}(s) \) represents the tables accessed by \( s \), \( \text{col}(s) \) represents the columns accessed by \( s \), and \( \text{op}(s) \) represents the operation (SELECT, INSERT, UPDATE, or DELETE) performed by \( s \). To compute the traceability between test cases and database entities, our approach traces the SQL commands that get issued during the execution of a test case. For each SQL command \( s \) that gets issued, the approach identifies the triggers associated with the tables that are accessed by \( s \).

More formally, the traceability between test cases and database entities is represented as \( [t, \{s\}, \{D_t\}] \), where \( t \) is a test case, \( \{s\} \) is the set of SQL commands that are issued during the execution of \( t \), and \( \{D_t\} \) is the set of triggers associated with \( \text{tab}(s) \).

Depending on the specific API used to execute the SQL commands, appropriate probes can be added to the program to track the commands issued during test execution.

### C. Test Selection

The final analysis uses the results of the differencing analysis and the traceability analysis to select the test cases that cover the affected properties and database entities.

1) **Test selection for properties:** Given an affected property \( r \), test selection uses the traceability information to identify the tests that cover \( r \). For the example in Figure 2, the modified property URL_INTERNAL has tests \( t_1 \) and \( t_3 \) associated with it. Therefore, those test cases are selected. If test selection were to be performed at the file-level (to select the tests that access the modified file), all four tests would be selected.

2) **Test selection for database entities:** For an affected database schema or trigger, test selection has to identify the test cases that cover the affected tables, columns, and triggers. First, we discuss test selection for affected tables; following that, we discuss test selection for affected triggers.

The traceability information for databases associates test cases with SQL commands. The test-selection algorithm parses each SQL command \( s \) to identify the tables and the table columns accessed by \( s \) and uses that information, along with information about the way \( T \) is affected, to select the affected tests. Let \( \sigma(s) \) be a function that maps an SQL command to the set of test cases that cover the command. If \( T \) is deleted, all the test cases that cover those SQL commands that access the table are selected.

\[
\{\sigma(s) \mid T \in \text{tab}(s)\}
\]

Note that a table deletion would typically be accompanied by changes in the code, for which the code-centric RTS techniques can be used to identify the affected tests. Therefore, our approach may not provide improved accuracy for schema changes in which tables are deleted. Similarly, the addition of a new table would be accompanied by code changes, which can be detected by the existing RTS techniques.

If the modification to table \( T \) alters the type of, or deletes, a column \( c \), the behavior of SQL commands that access \( c \) is affected. Thus, the selected tests are

\[
\{\sigma(s) \mid T \in \text{tab}(s) \land c \in \text{col}(s)\}
\]

A column-level test selection avoids selecting the test cases that access only the unmodified columns. This can make it more accurate than a table-level selection, which selects each test case that accesses the table, irrespective of the columns that are read.

In the case of database triggers, the deletion or modification of an existing trigger, or the addition of a new trigger, can modify the behavior of existing test cases. For an added or deleted trigger \( D_t = (T, e, A, p) \), each SQL command that satisfies the trigger event is affected; consequently, the test cases that cover those commands can behave differently with the new trigger. Therefore, for an added or deleted trigger, the selected tests are

\[
\{\sigma(s) \mid T \in \text{tab}(s) \land e = \text{op}(s)\}
\]

For an affected trigger \( D_t = (T, e, A, p) \), test selection again varies depending on the element of \( D_t \) that is changed. If \( T \) is changed to \( T' \), the selected tests are

\[
\{\sigma(s) \mid T \in \text{tab}(s) \land e = \text{op}(s)\} \cup \\
\{\sigma(s) \mid T' \in \text{tab}(s) \land e = \text{op}(s)\}
\]

Such a modification to a trigger, which causes the trigger to attach to a different table \( (T') \), affects the behavior of those SQL commands that perform the relevant operation \( e \) on \( T \). The execution of these commands would no longer
cause the trigger to be fired. Moreover, this modification also affects the behavior of the commands that perform operation $e$ on $T'$ because, with the modification, the execution of those commands would cause the trigger to fire, which did not occur in the previous version.

Similarly, if $e$ is changed to $e'$, the selected tests are

$$\{\sigma(s) \mid T \in \text{tab}(s) \land e = \text{op}(s)\} \cup \{\sigma(s) \mid T \in \text{tab}(s) \land e' = \text{op}(s)\}$$

This affects all SQL commands that perform the old operation $e$ on $T$ and the commands that perform the new operation $e'$ on $T$.

Finally, if $A$ or $p$ is modified, the selected tests are

$$\{\sigma(s) \mid T \in \text{tab}(s) \land e = \text{op}(s)\}$$

D. Limitations of Current Approach

Our current approach focuses on a simple model of property files and accommodates limited types of database entities. As we stated in the Introduction, the scope of research on the topic of regression testing in the presence of non-code changes is vast; this work is part of a larger ongoing project and focuses on a subset of the problem space. In this section, we highlight some aspects of the problem that we have not yet addressed, but intend to investigate in future work.

In our approach, the representation of properties as a collection of name-value pairs is adequate for plain-text property files. However, configurations can have a richer structure, typically specified in XML files. To illustrate with an example, consider the following XML fragment, which specifies two types of URLs, each with different attributes:

```xml
<url>
  <internal host="localhost" port="8080"/>
  <external host="ext.host.com" port="9080" locale="1033">
    <login user="anon" password="welcome" locale="1033"/>
  </external>
</url>
```

Such a property specification has a hierarchical structure that can be modeled as a tree. A non-leaf node in the tree represents an XML element (e.g., `<internal>`), whereas a leaf node represents an attribute (e.g., `port=8080`), which is a pair consisting of an attribute name and an attribute value. More generally, XML files can contain information, other than configurations, that affects the program behavior. In future work, we will extend our technique to handle this type of information; the extension is fairly straightforward and will mostly require an engineering effort.

Our current approach models the database schema and triggers; a more comprehensive model would consider other entities, such as stored procedures, views, and integrity constraints. A stored procedure is a subroutine, defined and stored in the database system, that can be accessed by applications interacting with the database. Stored procedures can be implemented in proprietary languages (e.g., PL/SQL for Oracle databases) or general-purpose languages (e.g., DB2 and Oracle let stored procedures to be programmed in Java), and typically contain a mix of procedural and declarative code. Changes in such code can affect the application behavior directly, if the application invokes the procedures, or indirectly, by affecting the behavior of triggers that invoke the procedures. Although we do not do it in the current formulation of our technique, code-based RTS techniques could be used to analyze changes to stored procedures, once we suitably accommodate the features of the proprietary languages (or the common idioms) used in stored procedures.

Another aspect of modeling databases is modeling their data content. Modifications in the data content can affect the application behavior and require retesting. Specific patterns in the data may also reveal faults that would not be revealed otherwise. The data content could be modeled in different ways. For instance, we could model the content at the level of groups of records or even individual records. Another option would be to not model specific data or sets, but rather to make statistics about the data and its distribution part of the model. One reason why we have not handled data content yet is that it is questionable whether a change in content constitutes a change in the application inputs or a change in the application environment. Depending on the context, either of these views may be appropriate. For now, we therefore chose not to consider the content and focus instead on the structure and functionality of a database alone.

IV. EMPIRICAL EVALUATION

To validate our technique, we implemented it in a prototype tool called TREDS (Tool for Retesting Environment-Dependent Software). The current version of TREDS can handle Java applications that interact with text-based property files and MYSQL databases. We then used TREDS to assess feasibility, effectiveness, and potential usefulness of our approach. To do this, we performed an empirical study in which we used TREDS on two real systems: a proprietary application and an open-source system. In the study, we compared our approach with a conventional code-centric RTS technique (e.g., [8]) in terms of the number of test cases they select for a given set of changes to the two subjects considered. In the rest of this section, we first describe our experimental setup and then present our results.
Table II

<table>
<thead>
<tr>
<th>Subject</th>
<th>Version Pair</th>
<th>Property Modification</th>
</tr>
</thead>
<tbody>
<tr>
<td>CA</td>
<td>(v0, v1)</td>
<td>Work-product cover page URL</td>
</tr>
<tr>
<td></td>
<td>(v0, v2)</td>
<td>Mapping file for artifact Variant</td>
</tr>
<tr>
<td></td>
<td>(v0, v3)</td>
<td>Layout file for artifact Business Benefit</td>
</tr>
<tr>
<td></td>
<td>(v0, v4)</td>
<td>Image service URL</td>
</tr>
<tr>
<td></td>
<td>(v0, v5)</td>
<td>Server IP URL</td>
</tr>
<tr>
<td></td>
<td>(v0, v6)</td>
<td>Location of folder</td>
</tr>
<tr>
<td></td>
<td>(v0, v7)</td>
<td>Tool instantiation type</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subject</th>
<th>Version Pair</th>
<th>Schema Modification</th>
</tr>
</thead>
<tbody>
<tr>
<td>iTrust</td>
<td>(v0, v1)</td>
<td>Trigger action: adds record</td>
</tr>
<tr>
<td></td>
<td>(v0, v2)</td>
<td>Trigger table: attaches to a different table</td>
</tr>
<tr>
<td></td>
<td>(v0, v3)</td>
<td>Trigger event: executes on UPDATE instead of SELECT</td>
</tr>
<tr>
<td></td>
<td>(v0, v4)</td>
<td>Column made non-null</td>
</tr>
<tr>
<td></td>
<td>(v0, v5)</td>
<td>Column made non-null</td>
</tr>
</tbody>
</table>

A. Experimental Setup

1) Subjects: We used as experimental subjects one proprietary application developed at IBM Research—Consultant’s Assistant (CA)—and one open-source application—iTrust. Table I provides information about these subjects. We used CA to evaluate our approach for changes to property files, whereas we used iTrust to evaluate our approach for database changes.

CA is a collaborative requirements-authoring tool for business consultants [15]. The tool supports harvesting, storing, and reusing project artifacts, such as process models, across business-consulting projects. CA follows a typical three-tier system architecture consisting of client-side scripts, back-end services, and data repositories; its components include Java, JavaScript, XSLT, HTML, XML, and plain-text property files. For this study, we analyzed the Java programs and the plain-text property files only.

The CA system supports the generation of 7 types of work products, such as process-definition documents and scenario documents, and 60 different types of artifacts, such as use cases and process steps. Business consultants use the system to capture client requirements, identify existing assets in the CA data repository that can be reused (as-is or with modifications) to meet a particular requirement, and automatically generate the required work products. The CA test suite currently consists of 108 test cases that attain 22% method coverage and 18% statement coverage.

To create versions, we used a base version $v_0$ of CA, and seeded different types of changes to a property file in that version to create seven modified versions $v_1$–$v_7$. The modified property file specifies values for different URLs, system paths, and so on. Some properties specify XML files that contain information about how a particular work product is generated. Table II presents information about the types of property changes made in each version. The seeded changes reflect realistic scenarios that could occur during the evolution of CA.

As stated above, to evaluate our approach for database applications, we used the open-source project iTrust, a medical web application developed at an academic institution (http://agile.csc.ncsu.edu/iTrust/wiki/doku.php). iTrust provides an interface to both patients and medical staff. It lets the patients communicate with their doctors and keep up with their medical history and records. It lets the medical staff keep track of the patients, schedule visits, prescribe medication, order and view lab results, and so on.

The iTrust distribution consists of a test suite of 842 tests, which attains 89% method coverage and 58% statement coverage. The distribution also comes with a MYSQL database with populated data content. The database schema consists of 36 tables, and the data content that comes with the distribution adds a total of 416 rows across all tables.

For this study, we intended to evaluate the effects of modifications to database triggers and schema. By seeding such changes into the base version $v_0$ of iTrust, we created five versions (see Table II). The iTrust database does not contain any triggers. Therefore, we created three triggers and seeded modifications to them: the modifications change the trigger table, event, and action. We also created two modifications in each of which we changed the is_null flag of a column from true to false.

2) Implementation: Our prototype tool, TREDs, implements the differencing, traceability, and test-selection analyses for Java properties and Java interactions with MYSQL databases that we described in Section III.

TREDs contains two differencing components: PROPDIFF, which computes the differences between two text-based property files, and DBDIFF, which computes the differences between two MYSQL databases. PROPDIFF compares two property files containing name-value pairs and identifies added, deleted, and modified properties. DBDIFF compares two MYSQL database instances and identifies deleted, added, and modified entities—either tables or triggers. For a modified trigger, the differencing identifies which trigger elements are changed, as discussed in Section III-A2.

The traceability analysis requires the association of test cases with properties, SQL queries, and the associated database triggers. To do this, TREDs includes an instrumenter developed using BCEL, the Byte Code Engineering Library (http://jakarta.apache.org/bcel). The instrumenter adds probes to Java classes to trace calls to part of the Java property API (java.util.Property) and to the Java database API (java.sql). The instrumenter also adds probes to track the execution of methods, branches, and exception handling. For properties, the added probes record the read property files and properties, and associate it with the currently executing test case. For database interactions, the probes record the issued SQL commands and also suitably associate them with test cases. A separate component analyzes the SQL coverage information to link each SQL command to the database triggers that are executed in response to the command.

Finally, TREDs contains a test-selection component that, given the results of the traceability analysis and information about the seeded changes, identifies the affected test cases.
B. Effectiveness of Test Selection

The goal of this study is to evaluate the effectiveness of our technique in selecting tests for changes to property files and database triggers. To do this, we compared the tests selected by our approach with the tests selected by a code-centric technique. We considered two versions of the code-centric technique: the basic version, with no extensions, and an enhanced version with a simple extension for accommodating changes in property files and databases. In the enhanced version, each program point at which a modified property file is loaded, or a connection to a modified database is established, is marked as modified.

Specifically, in this study, we investigated the following research questions:

- **RQ1:** To what extent can a code-centric RTS technique miss selecting the modification-traversing tests that are selected by our approach?
- **RQ2:** To what extent can the enhanced code-centric RTS technique select unnecessary tests (i.e., tests that are not modification-traversing) that are not selected by our approach?

Note that the enhanced version of the code-centric technique always selects a superset of the test cases selected by our technique. Thus, whereas RQ1 investigates the improved recall of test selection achieved by our approach, RQ2 investigates the improved precision attained by our approach over the simple extension to a code-centric technique.

1) Effectiveness for changes in properties: Figure 5 presents the data about the effectiveness of test selection for the seven CA versions. The horizontal axis represents the version pairs, and contains three bars for each pair, which represent the test cases selected using retest-all, the extended code-centric RTS technique, and our approach, respectively. The vertical axis represents the percentage of test cases in the test suite selected using each technique.

The bar on the right provides data to answer RQ1: it shows the percentage of test cases that are modification-traversing for the affected properties (i.e., the test cases that access the modified properties) that are missed by a code-centric technique. The percentage of missed test cases varies from 0%, for \((v_0, v_4)\), to 39%, for \((v_0, v_6)\). For \((v_0, v_4)\), none of the test cases in the test suite read the modified property; therefore, no test cases need to be rerun.\(^2\) Overall, the data illustrate that an RTS technique that does not accommodate changes to property files can miss many test cases.

The middle bar shows the percentage of the test suite that is selected by the enhanced code-centric technique. The enhanced technique selects each test case that accesses the modified property file, irrespective of whether a test case actually reads the modified properties. For this study, we modified the same property file for each version pair, and 43% of the test cases read that file. Therefore, for each version pair, these test cases are selected. The additional test cases selected by the code-centric analysis over the test cases selected by our technique represent the imprecision in the code-centric technique. These test cases are not modification-traversing, but, nonetheless, are selected. As the data show, the maximum imprecision occurs for \((v_0, v_4)\) for which all selected test cases are non-modification-traversing. On the other end of the imprecision range, for \((v_0, v_6)\), only 9% of the selected test cases are non-modification traversing.

Overall, the data indicates that the simple approach of selecting each test case that covers a statement at which a modified property file is read can be very imprecise in practice. On average, the imprecision is 72% for the seven versions of CA. However, this simple approach still achieves significant reduction—57% for each subject—over the retest-all testing strategy, which would rerun all test cases.

2) Effectiveness for changes in databases: Table III presents data to illustrate the effectiveness of our technique in selecting test cases for iTrust. As the data indicates, our approach attains significant reduction in the number of test cases selected. This occurs because the changes are very specific, and our approach (1) accurately analyzes the changes made to the database triggers and tables and (2) selects only the test cases that execute the affected SQL commands. A purely code-centric technique, however, would miss these test cases. The number of selected test cases is greater for \((v_0, v_4)\) and \((v_0, v_5)\) than the other version pairs. \(v_4\) and \(v_5\) contain changes to the is_null flag of columns, which affects all SQL commands that perform select, insert, or update commands involving the changed columns. Versions \(v_1, v_2, v_3\), and \(v_5\) contain changes to triggers that execute under more...

\(^2\)In such cases, the test suite has to be augmented to add test cases that would cover the modified properties. However, in this paper, we focus on test selection, and do not address the implications of modifications to property and database entities for test-suite augmentation.
restricted SQL commands and, consequently, the changes affect fewer SQL commands.

For database interactions, the enhanced code-centric RTS selects each test that interacts with the modified database. Of the 842 tests for iTrust, 536 perform database operations—all of these are selected by the enhanced code-centric RTS, irrespective of the type of change. The imprecision in the test cases selected by the enhanced RTS varies from 86% to over 99%. Because the database modifications are very specific, the approach of selecting all test cases does not perform well. The enhanced RTS still attains approximately 36% reduction over the retest-all strategy.

C. Threats to Validity

The most significant threats to the validity of our results are threats to external validity, which arise when the observed results cannot be generalized to other experimental setups. In our study, we used only two subjects, and we seeded modifications to create different versions. The performance of the technique may vary for other subjects and types of changes. Although the seeded changes reflect realistic scenarios for the subjects and are not arbitrary in nature, experimentation with real changes is necessary to validate our preliminary results. Our study was also limited to a simple model of property files and considered only two types of database entities. Extensions of the technique to consider other external entities and additional experimentation are required to assess the applicability and usefulness of our technique in practical scenarios.

V. Related Work

Regression testing is an active area of research that has accumulated a rich body of work. Researchers have presented many regression-testing techniques, including regression test selection (e.g., [4]–[10]), test-suite prioritization (e.g., [16]), test-suite augmentation (e.g., [17]), and test-suite reduction (e.g., [18]). However, most of these techniques are code-centric and ignore non-code changes in a system or changes in the environment of a system.

There also exist techniques that perform test selection by analyzing artifacts at higher levels of abstraction than code, such as architecture diagrams [13] and UML design models [14]. However, these techniques also do not consider changes in configuration files and databases.

Haraty and colleagues [19] present a technique for selecting tests cases in the presence of modifications to database entities. Their technique is applicable to stored procedures only: for each database command in a stored procedure that accesses a modified trigger or table, the technique selects the tests that cover that statement. The goal of our work is broader. First, our approach addresses changes to configuration files. Second, our technique targets general-purpose applications, such as web applications, which makes it applicable to a large class of modern software systems. Finally, general-purpose applications present analysis challenges that stored procedures do not. For example, stored procedures contain explicit SQL commands, which can be identified directly using a syntactic analysis, whereas general-purpose database applications frequently construct database queries dynamically. Our approach of recording SQL queries at runtime and creating traceability between test cases and SQL commands enables an accurate analysis in the presence of such dynamic SQL construction.

Willmor and Embury [20] present a safe test-selection technique for database-driven applications. Their technique considers the scenario where a code change affects a database-interacting statement, such that the statement’s effect on the persistent database state changes (which can in turn affect the behavior of other statements that read the modified persistent state). To ensure safe test selection, their approach identifies statements that are dependent (through persistent state) on modified programs parts and marks them as affected. Unlike their approach, our approach is applicable in cases when there is no change in the code, but changes in property files and database entities, such as triggers, affect software behavior.

Other research has addressed related, but orthogonal, problems, such as improving the execution efficiency of regression tests by reducing the amount of database resets and executing tests in parallel (e.g., [21]), defining coverage criteria for database applications (e.g., [22]), monitoring database coverage by test cases (e.g., [23]), and developing frameworks for testing database applications (e.g., [24]).

VI. Summary and Future Work

In this paper, we presented a new RTS technique that performs accurate regression test selection in the presence of environment changes (specifically, changes to property files and databases) that can affect software behavior. By doing so, our technique overcomes the limitation of existing RTS techniques, which focus mainly on code changes. The use of configuration files and databases is pervasive in modern software systems. Therefore, to be applicable and effective in practice, an RTS technique must accommodate changes to those components, as our technique does. We also presented preliminary empirical results that indicate that code-centric RTS can fail to select many test cases that are affected by configuration and databases changes. Our results also show that a simple approach that treats a property file or a database as the changed entity, and does not analyze accurately the specific changes and their effects, can select a large number of unnecessary tests.

The broader goal of our work is to develop novel regression-testing techniques that are applicable for, and effective in, analyzing modern software systems that are characterized by heterogeneity and environment dependence. This paper represents the first step in this vast research area—addressing environment-dependence. In the near future, we
will extend our technique to accommodate more complex external entities. We will also create appropriate testbeds, with varied and realistic subjects, which will let us perform further and more systematic experimentation. In the longer-term, we will address the heterogeneity aspect of modern software systems and consider additional environment dependencies, such as network layouts, that complicate regression testing.

ACKNOWLEDGEMENT

This work was supported in part by NSF awards CCF-0541049, CCF-0725202, CCF-0916605, and CCF-0964647 to Georgia Tech.

REFERENCES