A Model-based Approach to Regression Testing of Component-based Software *

Chuanqi Tao, Bixin Li
School of Computer Science and Engineering
Southeast University, Nanjing, China
chuanqi.tao@sjsu.edu, bx.li@seu.edu.cn

Jerry Gao
School of Computer Engineering
San Jose State University, San Jose, USA
gaojerry@email.sjsu.edu

Abstract—Component-based software systems consist of various components, such as third-party components and in-house built components. Due to the component changes, a software system is usually affected at both component level and system level. Related existing research does not address the issue of systematic regression testing of component-based software, especially at system level. This paper proposes a systematic regression testing method from components to system. The paper discusses component API changes, interaction changes, and architecture changes. In addition, it presents a component-based change impact analysis method based on the proposed component firewalls and uses a decision table as its test model. The provided approach is applied throughout the regression testing process. Finally, the paper reports our case studies based on a realistic component-based software system. The study results show that the approach is feasible and effective.

Keywords-component-based software regression testing; software maintenance; retest model; change and impact analysis; test cases update

I. INTRODUCTION

Component-based software is widely used nowadays. The modern software system is primarily constructed based on reusable components, such as third-party components and in-house built components. During software maintenance, when a component is updated or upgraded, it must be retested. This refers to regression testing, which is an important task of software maintenance. Its main objective is to gain the quality confidence for the updated software whenever it is changed. According to [1], regression testing is a major task of software maintenance and it accounts for more than one-third of its total costs. For any component-based software, its regression testing can be conducted in a hierarchical manner that is from the component-level unit retest, component reintegration, to the system level regression testing.

In regression testing of component-based software, the research topics still focus on re-test model, change impact analysis and test case update. However, changes made to a component could bring impact on the other parts of the component, which means the change impact could affect other components of the system or the whole system behaviors. In addition, component-based regression testing should take practical test models into account. For instance, if a decision table-based method has been selected as a test model to generate test cases, regression testing should consider how to identify test cases change and impact based on the decision table.

Research in the past seldom discussed how to identify component changes and impacts in component-based software. Although some papers discussed regression testing of component [2–5], these papers didn’t address regression testing at different levels from component to system. Nowadays, practitioners in the real world are looking for systematic solutions to support component and system regression testing and component evolution.

This paper addresses those needs above by providing a systematic approach to regression testing of component-based software based on re-test models. Re-test models are usually used to present the dependency relationships amongst components, assist engineers to define re-test criteria and re-integration strategies, and facilitate automatic test generation. To support regression testing, related modes need to be chosen. This paper proposes several models according to diverse views of component testing. In those models, the relationships between functions or data at component level, between components at system level are all taken into account.

Change identification is the first step of regression testing. A clear classification of change types can support effective impact analysis. The various change types are summarized at different levels in this paper. Impact analysis is an important task in regression testing. The component firewall concept is utilized as a primary method for our change impact analysis. This paper presents diverse firewalls based on the re-test models at both component level and system level. Test case update is one of the goals of regression testing. As a complete regression testing solution, the original test cases should be updated for reusable test cases, obsolete test cases and new test cases. To identify the affected test cases, the affected components or parts should be mapped into the corresponding test cases. Thus, the test models which are used for generating test cases need to be analyzed. Since the decision table-based testing is an important component testing method and the project of our case study also utilizes decision table as the major testing method, therefore, we regard decision table as a basic test model for our regression testing in this paper. The related test case update is based on the decision table. Therefore, the process to perform regression testing of component-based software from component to system is as following steps:

Step 1 Change identification at component level, which refers to API function changes, API data changes, structural changes, etc.

Step 2 Impact analysis at component level, which refers to component function firewall, component data function firewall and component API function firewall.

Step 3 Test cases update at component level, which refers to component decision table-based test cases reused, deleted and added.

*Supported partially by the National Natural Science Foundation of China under Grant No. 60773105 and no. 60973149, and partially Supported by Doctoral Fund of Ministry of Education of China under Grant No. 20100092110022, and partially by National High Technology Research and Development Program under Grant No. 2007AA01Z141 and No. 2008AA01Z113.
Step 4 Change identification at system level, which refers to component interaction changes and architecture changes.
Step 5 Impact analysis at system level, which refers to extended composition tree firewall and component interaction firewall.
Step 6 Test cases update at system level, which refers to system decision table-based test cases reused, deleted and added.

The major contributions of this paper are summarized below:
(1) A systematic solution to regression testing of component-based software from component level to system level is presented, including the process of regression testing: change identification, change impact analysis and test cases update.
(2) Several new component firewalls are introduced for component change impact and test case reuse in component-based software regression testing.
(3) Practical application experiments are performed in our case studies. Nearly a hundred students joined related empirical studies.

This paper is organized as follows. Section 2 introduces a brief review of the related work in regression testing of component-based software. In section 3, the re-test models are presented. Section 4 analyzes the change types and provides a systematic approach to component-based software change impact analysis using firewall. Section 5 discusses test cases update. Section 6 reports the case study. Conclusion and future work are summarized in the end.

II. RELATED WORK

In the past decades, a number of papers have been published for regression testing issues for conventional programs and object-oriented software. Those papers primarily focus on three issues: regression test selection techniques [6, 7, 9, 10], regression testing cost-effectiveness analysis [11, 12], and object-oriented re-integration [13–15].

A lot of papers focus on the component-based software testing issues. Currently, component-based software testing mainly focuses on component testability, component test adequacy and coverage, component-based software integration, performance testing, and configuration testing. Recently, a few papers addressed the regression testing problems existed in component-based software [2, 3, 16–18]. They can be generally classified into the following three groups.

The first group is regression test selection of component-based software. For example, Harrold et al. proposed an approach to regression testing of COTS components using component metadate [16]. They utilized three types of meta-data to perform the regression test selection. However, the method needs additional information from component which may be not available in practice. Similarly, Orso et al. also discussed two techniques for regression testing of component-based software [2]. The first is code-based and the second is specification-based. Both techniques are based on the provided component meta-data. To support the approach, the additional information is needed, including the version information, change data and coverage measurement facilities. Zheng et al. proposed an Integrated-Black-box Approach for Component Change Identification for COTS(Commercial-off-the-shelf) software [17]. For the third-party component, the internal software information could be available from component specification, user interface and reference manual. To support the approach, binary code and documentation should be visible. They assumed that when components change and only binary code and documentation are available, regression test selection can safely be based upon the glue code that interfaces with sections of the component that changed. Robinson et al. proposed a firewall method for regression testing of user-configurable software. They constructed a firewall to identify the impacted area in system based on setting changes and configurable element changes respectively, then created or selected test cases to cover the impacts [19].

The second group is UML-based. For instance, Wu et al. presented a UML technique for regression testing of component-based software [4]. They adopted UML diagrams, which represent changes to a component, to support regression testing. Class diagrams, Collaboration diagrams, and Statechart diagrams are considered to be as the re-test models.

The third group is the systematic method based on API models. Gao et al. focused on component API-based changes and impacts, and proposed a systematic re-test method for software components based on a component API-based test model [3]. In addition, Mao et al. proposed an improved regression testing method based on built-in test design for component-based system [5].

Open questions and challenges of regression testing of component-based software are primarily as follows.

-How to identify the diverse component changes in a systematic way?
-How to do impact analysis from component to system?
-How to update both component and system test cases?

This paper addresses those problems above. Unlike previous work that only focused on component change analysis and impact at the component level, this paper proposes a model-based approach for regression testing. A firewall approach is utilized for impact analysis from component to system to find out ripple effects on other components and system behaviors. Moreover, this paper also introduces a systematic methodology for regression testing of component-based software from component to system.

III. COMPONENT-BASED SOFTWARE RE-TEST MODELS

Component and system can be viewed from different perspectives for testing. For instance, a component can have white-box view, black-box view, API view or performance view. A system can have integration view, configuration view, function view or performance view. Component testing researchers can choose different views according to their test plan and test goal. Various views of components correspond to different re-test models.

At component level, the changes could be structure changes or internal logic changes. If each updated component is assumed to provide the component internal information as its meta-data, from the white-box view, the white-box re-test models like data function dependency or function dependency could be adopted for component re-test. Since component function is usually used through API function or data call, the API models can be borrowed for component re-test. From the black-box view, the component could be tested using decision table-based testing or state-based testing. Hence, the related test models could be adopted for component testing. In addition, from the performance view, the scenario-based testing models can also be applied. However, performance is out of the scope of this paper.

At system level, the relationship between components could be interaction, composition, message communication, etc. Thus, re-
lated system-level models are proposed for system regression testing. For example, from the integration view, component interaction graph can be used as interaction models for system. As component system is usually configurable, the composition and configuration models are needed to support the system testing. From the function view, decision table-based or state-based methods could also be adopted for system function testing.

The reason why we adopt those models to support the regression testing of component-based software from component to system is explained above. In the next few subsections, we will define and introduce those models in detail.

A. System-level Re-test Models

At the system level, from the integration view and configuration view, we propose two re-test models. Now we introduce those two models in detail.

1) Extended Composition Tree: Extended Composition Tree (ECT) describes the composition and configuration relationship amongst components or inside sophisticated components. ECT extends traditional composition tree through adding the configuration relationship [20]. For instance, an elevator system is composed of the car, the floor panel, the controller, and etc. Sophisticated component car contains user panel, door, car controller, etc. Regarding configuration, a door component can be configured with Single door or Double door.

ECT consists of tree nodes and links. Tree nodes present single or sophisticated components including configurable components. Tree links in ECT present the composition relation or configuration relation between tree nodes.

Definition 1 An Extended Composition Tree (ECT) can be defined a directed graph $E = (N,E,R)$, where $N = \{N_1,N_2,...,N_n\}$ is a finite set of nodes, $R = \{POW, EXT, EOR, AND, Switch, Multiplex\}$ is the set of relations, and $E = E_{POW} \cup E_{EXT} \cup E_{EOR} \cup E_{Multiplex} \cup E_{AND} \cup E_{Switch}$ is the set of edges. $E_{POW}$ is the set of directed edges representing the part-of-whole relation between the components. For any two components $C_1,C_2 \in N$, $(C_1,C_2,POW) \in E_{POW}$ indicates that component $C_2$ is a part of $C_1$, $E_{EXT}$ represents the set of directed edges representing the extend relation between the components. For any two components $C_1,C_2 \in N$, $(C_1,C_2,EXT) \in E_{EXT}$ indicates that component $C_2$ extends $C_1$ through composition.

Regarding configuration relation, we already defined the related models in our previous work [20].

2) Component Interaction Graph: Wu et al. introduced the component interaction graph to depict interactions and dependance relationships among components, which is mainly call relationship [21]. Here, Component Interaction Graph (CIG) addresses the relationships include message-communication, usage, etc. Component interaction graph is defined as Definition 2 below.

Definition 2 Component interaction graph (CIG) for software components is a directed graph $CIG = (N,E,R)$, where $N$ is the set of nodes representing the components, $R = \{MSG,USA\}$ is the set of interaction relations, and $E = E_{MSG} \cup E_{USA}$ is the set of edges defined below. $E_{USA} \subseteq N \times N \times R$ is the set of directed edges representing the usage interaction relation between the components. $E_{MSG} \subseteq N \times N \times R$ is the set of directed edges representing the message-based interaction relation between the components.

![Figure 1. A Sample of Component Semantic Decision Table](image)

For instance, For any two components $C_1,C_2 \in N$, $(C_1,C_2,MSG) \in E_{MSG}$ indicates that component $C_1$ sends message to $C_2$.

B. Re-test Models for Test Cases Reuse

1) Component Semantic Decision Table Test Model: Decision Table is a black-box testing method focusing on validating business rules, conditions, constraints and corresponding responses and actions of a software component [22, 23]. The basic approach is to identify and list all possible conditions and their combination cases as well as responding actions and outputs for each case, then define test cases to cover each case. In this paper, we propose a new Component Decision Table for component. We call this Component Semantic Decision Table (CSDT). For each case, there exists precondition, action and postcondition.

According to component black-box view, the data value of preconditions could be Incoming data, Incoming message(such as GUI input data or component internal data) or API call parameter, and the data value of postcondition could be Outgoing data(such as database table, GUI output, output data file, etc.), Outgoing message or Outgoing call data. All these preconditions and postconditions could be obtained from component specification. Action in CSDT stands for component function. Figure 1 shows a sample component semantic decision table. The precondition, postcondition and action are explained in Definition 3.

Definition 3 The CSDT includes three sets, which is presented below. $prec = \{prec_1,prec_2,...,prec_n\}$, $postc = \{postc_1,postc_2,...,postc_n\}$, where $prec$ denotes a precondition set, which includes precondition $prec_1,prec_2,...,prec_n$ and postc denotes a postcondition set, which includes postcondition $postc_1,postc_2,...,postc_n$, action $= \{A_1,A_2,...,A_i\}$ where action denotes a action set, which includes action $A_1,A_2,...,A_i$.

2) System Semantic Decision Table Test Model: In our component system specification, there is a feature-component table. The table describes the relation between system features and corresponding components. These features can be observed at the system level. For each feature, there is a corresponding decision table.

The system features can be defined as a set $Ft$, $Ft = \{ft_1,ft_2,...,ft_n\}$, where $ft$ denotes the function feature set of component-based system, which includes feature $ft_1,ft_2,...,ft_n$. Each feature is supported by a set of components. The relation between feature and components can be expressed below.

$$ft_i \rightarrow C(ft_i) = \{C_1,C_2,...,C_m\}$$

Each system decision table can be used for each feature, which is presented below.

$$ft_i \rightarrow SDT_i$$, where $SDT_i$ stands for system decision table.

$$SDT_i \rightarrow STd_i$$, where $STd_i$ stands for test case set for decision table $SDT_i$. 
Table: Component Function and Interaction Change Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Specific Changes</th>
<th>Type</th>
<th>Specific Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADP</td>
<td>Add a data parameter</td>
<td>AMO</td>
<td>Add Message-based (MB) relation (outgoing)</td>
</tr>
<tr>
<td>AF</td>
<td>Add a function</td>
<td>AMI</td>
<td>Add MB (incoming)</td>
</tr>
<tr>
<td>ARD</td>
<td>Add a ‘return’ data</td>
<td>AMB</td>
<td>Add MB (both way)</td>
</tr>
<tr>
<td>DDP</td>
<td>Delete a data parameter</td>
<td>AU</td>
<td>Add Usage relation</td>
</tr>
<tr>
<td>DF</td>
<td>Delete a function</td>
<td>DMO</td>
<td>Delete MB (outgoing)</td>
</tr>
<tr>
<td>DRD</td>
<td>Delete a ‘return’ data</td>
<td>DMII</td>
<td>Delete MB (incoming)</td>
</tr>
<tr>
<td>CDT</td>
<td>Change data type or name</td>
<td>DMII</td>
<td>Delete MB (both way)</td>
</tr>
<tr>
<td>CFL</td>
<td>Change the function logic</td>
<td>DU</td>
<td>Delete Usage relation</td>
</tr>
<tr>
<td>CFS</td>
<td>Change the interface signature</td>
<td>CMS</td>
<td>Change MB semantic</td>
</tr>
</tbody>
</table>

Figure 2. Component Function and Interaction Change Types

Figure 3. Architecture Change Types

IV. CHANGE IMPACT ANALYSIS

We have summarized the common changes existed in component-based software in Figure 2 and 3. Component changes could be classified at two levels: a) the component level, and b) the system level. Change impact analysis is based on the models proposed in section 3. We introduce several component change firewalls, which include change, add and delete firewalls. The basic procedure to perform a component change impact consists of the following steps:

1) Identify the impacts of a changed component function or data on other component functions at component level; 2) Identify the component change impacts on its API; 3) Identify the component change impact on its precondition, action, and postcondition; 4) Identify the component change impacts on its API; 3) Identify the component change impact on other components based on interaction at system level; 5) Identify the component change impact on other components based on architecture at system level.

For any component, we assume $ECT = (N, E, R)$ be the old version of its Extended Composition Tree, $ECT' = (N', E', R')$ be its new version, $CIG = (N, E, R)$ be the old version of its Component Interaction Graph, and $CIG' = (N', E', R')$ be its new version.

A. Component function, precondition and post-condition Firewall

Through the computation of $CFFW$ and $CDFW$, we can get the affected component functions based on invocation dependencies and data define-use dependencies [3]. Various change types correspond to different impact. Now we try to present the change impact analysis corresponding to the summarized component level change types in Figure 2 using firewall.

- For ADP or ARD (the changed data is assumed as $Di$):
  \[ CDFW_{delete}(Di) = \{ F_j | (3F_j)(\exists F_i((F_i, F_j, du) \in R'_d - R_d) \land (F_i \in F) \land (\langle F_i, F_j, du \rangle \in R'_{FDG})) \} \]
  Where $R'_d$ is the data define-use relation derived from DFDG'.

- For AF (the changed function is assumed as $Fi$):
  \[ CFFW_{delete}(Fi) = \{ F_j | (3F_j)(\exists F_i((F_i, F_j) \in E' - E) \land (F_i \in F) \land (\langle F_i, F_j \rangle \in R'_f)) \} \]
  Where $R'_f$ is the dependence relation for FDG'.

- For DDP and DRD (the changed data is assumed as $Di$):
  \[ CDFW_{update}(Di) = \{ F_j | (\exists F_i((F_i, F_j, du) \in R_d - R'_d) \land (F_i \in F) \land (\langle F_i, F_j, du \rangle \in R'_{FDG})) \} \]
  Where $R_{d*}$ is the transition closure of $R_d$, which is the binary relation that define the data function-update relation between the residual component functions. $R_{d*}$ can be defined as:
  \[ R_{d*} = R_d \cap (F \times F \times \{du\}) \land (F \times F' \times \{du\}) \]

- For CDT (the changed data is assumed as $Fi$):
  \[ CDFW_{update}(Fi) = \{ F_j | (\exists F_i((F_i, F_j) \in R_{d*}) \land (\langle F_i, F_j \rangle \in R_{FDG})) \} \]
  Where $R_{d*}$ is the transition closure of $R_d$, which is the binary relation that define the dependencies between the residual component functions. $R_{d*}$ can be defined as:
  \[ R_{d*} = R_d \cap (F \times F \times \{du\}) \land (F \times F' \times \{du\}) \]

Regarding API function firewall, our previous work already discussed it [3]. The definition is as follows.

\[ CAW_{API}(C) = \{ F_i | (\forall F_i ((F_i \in F_{API} - F_{API}) \land (F_i \in CDFW(C)) \lor (F_i \in CDFW(C)))) \} \]

Where $CAW_{API}(C)$ includes all API functions which may be affected by component function changes, deletions and additions, as well as alters in the data-define-use relations between them.

Now we extend $CAW_{API}(C)$ by adding the precondition, post-condition, and action. Here, action corresponds to API function. precondition and postcondition change information can be obtained from the component requirement. The new firewall is called $ECAW$ (extended component API function firewall). Thus, the new firewall set can be represented as below.

\[ ECAW(C) = \{ (pre_{ci}, post_{ci}, action_{ci}) | (pre_{ci} \text{ is changed, added or deleted}) \lor (post_{ci} \text{ is changed, added or deleted}) \lor (action_{ci} \in CAW_{API}(C)) \} \]

Where $pre_{ci}, post_{ci}, action_{ci}$ denote precondition, postcondition and action respectively. $(pre_{ci}, post_{ci}, action_{ci})$ stands for a vector set, which include the affected preconditions, postconditions and actions.

B. Extended Composition Tree Firewall

Extended Composition Tree Firewall (ECTF) in Extended Composition Tree after changing components or configurations, refers to a set of components which might be affected by changing, adding and deleting components or configurations based on composition.
and configuration dependencies. ECTF is performed on architecture. Thus, it is used for identifying the affected architecture at the system level. For instance, in the elevator system, if the door component is changed, then the car component, which has the composition relation with door, will be affected. The affected test cases corresponding to this change mainly refers to unit test cases.

The Extended Composition Tree Firewall can be computed based on Extended Composition Tree. Now we introduce changed, added and deleted extended composition tree firewalls respectively.

The firewall for a changed, added, and deleted component $C_i$ can be computed below. $$ECTF_{\text{change}}[C_i] = [(C_i)|C_i \in C] \land ((C_i, C_j) \in R_{ECTF})$$ Where $R_{ECTF}$ is the transitive closure of $R_{ECTF}$, which is the binary relation that defines the composition dependencies between the components. $R_{ECTF}$ can be defined as follows:

$$R_{RECT} = R_{ECT} \cap (N \times N) \cap (N' \times N')$$

Where $\times$ is the Cartesian product operation.

$$ECTF_{\text{add}}[C_i] = [(C_i)|(3C_i)(3C_i)((C_i, C_i) \in E' - E) \land (C_k \in N) \land ((C_j, C_j) \in R_{ECT}))$$

Where $R_{ECT}$ is the composition dependence relation for ECT.

$$ECTF_{\text{delete}}[C_i] = [(C_i)|(3C_i)(3C_i)((C_i, C_i) \in E - E') \land (C_k \in N) \land ((C_j, C_j) \in R_{ECT}))$$

Now we present the change impact analysis corresponding to the summarized architecture change types in Figure 3 using firewall. The left side of arrow represents change type set and the right side of arrow denotes the corresponding firewall.

$$(APW, AE, ACDV, ACDT, ACF) \rightarrow ECTF_{\text{add}}[C_i]$$

$$(DPW, DE, DCDV, DCDT, DCF) \rightarrow ECTF_{\text{delete}}[C_i]$$

$$(CCDV, CCDT, CCF) \rightarrow ECTF_{\text{change}}[C_i]$$

For instance, in the elevator system, after adding an indicator in the car component and the floor panel component and a new protocol, the extended composition tree firewall (ECTF) is shown in highlighted in Figure 4.

C. Component Interaction Graph Firewall

Component Interaction Graph Firewall (CIGF) in Component Interaction Graph after changing components, refers to a set of components which maybe affected by changing, adding and deleting components based on interaction dependence. The Component Interaction Graph Firewall can be computed based on given Component Interaction Graph. For instance, in the elevator system, after adding an indicator in the car component and the floor panel component, the component interaction graph firewall (CIGF) is shown in highlighted in Figure 5. However, if we only consider component changes, the size of the Component Interaction Graph Firewall could be very large. Some components might be redundant for regression testing. API is the function interaction for other components to call. The Algorithm is shown in Algorithm 1. In the case of the interaction change types in Figure 2, the firewall could be computed by algorithm of Component Interaction Graph firewall.

![Figure 4](image1.png) Figure 4. A Sample Extended Composition Tree Firewall in the Elevator System

![Figure 5](image2.png) Figure 5. A Sample Component Interaction Graph Firewall in the Elevator System

**Algorithm 1 Component Interaction Graph Firewall**

Declare: $C_i$: changed component; CIG: component interaction graph; CIG: modified component interaction graph; CIGAPI$(C_i,F_i,CIG, CIG')$: Component interaction graph firewall CIGAPI$(C_i,F_i,CIG, CIG')$

```latex
\{\\%
\text{switch (change type)}\\%
\text{case 'Add functions':}\\%
CFFW\_\text{add}[C_i,F_i];
CFFW\_\text{add}[F_i,C_i];
S_f = CFFW\_\text{add}[C_i,F_i]\cup CDFW\_\text{add}[C_i,F_i]; //function firewall inside C_i
C_F = ECAW[C_i,F_i]; //API function firewall
break;
\text{case 'delete functions':}\\%
CFFW\_\text{delete}[C_i,F_i];
CFFW\_\text{delete}[F_i,C_i];
S_f = CFFW\_\text{delete}[C_i,F_i]\cup CDFW\_\text{delete}[C_i,F_i];
C_F = ECAW[C_i,F_i];
break;
\text{case 'change functions':}\\%
CFFW\_\text{change}[C_i,F_i];
CFFW\_\text{change}[F_i,C_i];
S_f = CFFW\_\text{change}[C_i,F_i]\cup CDFW\_\text{change}[C_i,F_i];
C_F = ECAW[C_i,F_i];
break;
\text{break;}
Mark each function in C.F visited; put C.F in CIGF; C_i = CIG[C_i]\text{link;} // (C_i,C_i) \in R_{CIG}
While ( C_i,f_i in C.F ) do
\{\\%
\text{if (C_i,F_i,f_i) \in P(C_i) \land C_i,F_i\text{not visited then}}\\%
CIGAPI[C_i,F_i,CIG, CIG']
\}\\%
\}
```
V. Test Cases Update

The final goal of regression testing is to refresh the existing test cases for the previous version, which means selecting reusable test cases and scripts, deleting out-of-date test cases and scripts, and adding new test cases and scripts.

As we mentioned above, decision table based testing is the primary method to generate test cases in this paper. Now we need to analyze how to map the change impact firewall into affected test cases in the given test suite. We introduce the concept of test firewall to emphasis the test cases updating. The procedure to perform test firewall analysis can be divided in two steps: 1) Identify the change impacts on component decision table-based test cases at component level; 2) Identify the change impacts on system decision table-based test cases at system level.

A. Decision table-based test firewall at component level

In any of the given component semantic decision table (CSDT), if any of the preconditions, postconditions and actions are added, deleted or changed, then the corresponding test cases could be affected.

According to the requirement changes, we can identify the component semantic decision table (CSDT) changes easily. As the test model defined in section 3, the CSDT changes could come from three sets: precondition, action and postcondition. The problem here is how to identify the reused, changed and new test cases according to the changes.

Each entry value in a row of a given decision table CSDT could be "True", "False", or "-". "True" means the corresponding precondition or postcondition has to be true; "False" means the corresponding precondition or postcondition has to be false. "-" means the corresponding postcondition could be either true or false. The table includes three sets, which is presented below. The entry value of precondition and postcondition can be defined as a set respectively like below.

\[
vprec = \{vprec_1, vprec_2, ..., vprec_m\}, \text{ where } vprec \text{ denotes the entry value set of precondition.}
\]

\[
vpostc = \{vpostc_1, vpostc_2, ..., vpostc_n\}, \text{ where } vpostc \text{ denotes the entry value set of postcondition.}
\]

Assumptions The value of precondition associated with test cases could be \(vprec(i \leq m) = "T" \lor "F" \lor "-"\). The value of postcondition associated with test cases could be \(vpostc(j \leq n) = "T" \lor "F" \lor "-"\). Assuming \(T_p \rightarrow \langle vprec, A, vpostc \rangle\), where \(T_p\) stands for decision table based test case. Vprec, \(A(A \subseteq \text{action})\) and vpostc stand for the associated precondition prec, action \(A\) and postcondition postc respectively.

Through the computation of ECW, we can get the firewall at component level. Assuming \(CTd_1\) denotes the original component decision table test cases set, now we need to get the updated version \(CTd_2\). According to the assumptions, we have the following rules for component decision table test firewall.

**Rule 1 (corresponding to change firewall):**

For any i, j, k

1. If \((\langle vprec, A, vpostc \rangle \in ECW_{change}(C)) \land (T_p \rightarrow vprec_i = "T" \lor vpostc_j = "T") \lor (T_p \rightarrow A_k \not\in ECW_{change}(C))\)

then \(T_p\) need to be changed in \(CTd_2\).

(2) If \((\langle vprec, A, vpostc \rangle \in ECW_{add}(C)) \land (T_p \rightarrow vprec_i = "-" \land vpostc_j = "-")) \lor (T_p \rightarrow A_k \not\in ECW_{add}(C))\)

then \(T_p\) need to be added in \(CTd_2\).

**Rule 2 (corresponding to add firewall):**

For any i, j, k

1. If \((\langle vprec, A, vpostc \rangle \in ECW_{add}(C)) \land (T_p \rightarrow vprec_i = "T" \lor vpostc_j = "T") \lor (T_p \rightarrow A_k \not\in ECW_{add}(C))\)

then \(T_p\) need to be added in \(CTd_2\).

2. If \((\langle vprec, A, vpostc \rangle \in ECW_{delete}(C)) \land (T_p \rightarrow vprec_i = "-" \land vpostc_j = "-")) \lor (T_p \rightarrow A_k \not\in ECW_{delete}(C))\)

then \(T_p\) need to be deleted in \(CTd_2\).

**Rule 3 (corresponding to delete firewall):**

For any i, j, k

1. If \((\langle vprec, A, vpostc \rangle \in ECW_{delete}(C)) \land (T_p \rightarrow vprec_i = "T" \lor vpostc_j = "T") \lor (T_p \rightarrow A_k \not\in ECW_{delete}(C))\)

then \(T_p\) need to be deleted in \(CTd_2\).

(2) If \((\langle vprec, A, vpostc \rangle \in ECW_{delete}(C)) \land (T_p \rightarrow vprec_i = "-" \land vpostc_j = "-")) \lor (T_p \rightarrow A_k \not\in ECW_{delete}(C))\)

then \(T_p\) need to be deleted in \(CTd_2\).

B. Decision table-based test cases firewall at system level

At the system level, we still adopt decision table model to perform test case update. In component-based system, there is a feature-component table, which describes the relation between system feature and corresponding components. These features can be observed at the system level. For each feature, there is a corresponding decision table. Here, the proposed firewall like component interaction graph firewall could be used to compute the affected components at the system level.

First we need to analyze if the changes affect the corresponding components. If so, then we find out the affected system features corresponding to the affected components. According to the relation between features and decision table, the affected decision table at feature level can be identified. Then, we analyze the changes to precondition, postcondition and action of decision table, and identify the test cases firewall at system level.

According to the System Semantic Decision Table Test Model, we propose the following formulas to compute the test cases firewall at system level. Here, \(C_0\) denotes modified component. CIGF and ECTF represent related change impact firewalls which are introduced above. Assuming the original system test cases set is \(STD_1\), we need to obtain the updated version \(STD_2\).

\[
STD_{create} = \{STD_i|\forall C_k(0 \leq k \leq m) \in C(f_1)\} \in \text{CIGF}_{create}(C_0) \vee \text{ECTF}_{create}(C_0))
\]

Where \(STD_{create}\) could be reused in \(STD_{2}\). \(C_0\) denotes modified component. \(C(f_1)\) denotes the set of components that support feature \(f_1\). \(STD_i\) represents the decision table corresponding to feature \(f_i\). The number of total components is \(m\).

\[
STD_{delete} = \{STD_i|\forall C_k(0 \leq k \leq m) \in C(f_1)\} \in \text{CIGF}_{delete}(C_0) \vee \text{ECTF}_{create}(C_0))
\]

Where \(STD_{delete}\) could be deleted in \(STD_2\).
\[ STd_{\text{change}} = |STd|([\exists C_k (k \leq m) \in C(f_1))(C_k \in CIGF_{\text{change}}(C_o) \lor ECTF_{\text{read}}(C_o))]
\]
Where \( STd_{\text{change}} \) need to be changed in \( STd_2 \).

\[ STd_{\text{new}} = |STd|([\forall C_k (k \leq m) \in C(f_2))(C_k \in CIGF_{\text{add}}(C_o) \lor ECTF_{\text{read}}(C_o))]
\]
Where \( STd_{\text{new}} \) need to be added in \( STd_2 \).

VI. CASE STUDY REPORT

To better understand our approach, we have performed a case study by applying the systematic regression testing from component level to system level onto a real component-based elevator system. We have used two software testing classes and two master project teams in San Jose State University (SJSU) to perform the related experiments. To make the study more typical, we conducted a complete component-based software testing process, including the original test cases design, test strategy, test coverage, etc. The decision table-based test cases are designed for both component and system. All of the test cases were executed adequately.

A. Study Objectives

The case study focuses on the following items: (a) Perform a systematic regression testing of the new component system version using the proposed approach, to verify the feasibility of the approach; (b) Check the effectiveness of the proposed approach; (c) Discover bugs after regression testing.

B. Study Subject

We have performed some case studies by applying the proposed approach in a component-based software, which is a component-based elevator system. The elevator system consists of several components, which are car, user panel, door, door panel, userpanel queue, car controller, floor panel and metatroller. We have used two software testing classes and two master project teams in San Jose State University (SJSU) to perform the related experiments. The test cases are decision table-based. In the new version, we have made some changes such as adding a component ‘Indicator’ in the component ‘Car’ to show the current floor where the car locates, adding a component ‘Indicator’ in the component ‘Floor Panel’ to show the current floor where the car locates, adding another kind of elevator algorithm to current system, like FCFS, SCAN, etc., to make elevator scheduling configurable, and so on. The original version of the system is well designed with adequate decision table-based component test cases and system test cases. In the new system version, we conducted regression testing from component level to system level, to obtain updated decision table-based test cases, including reused test cases, deleted test cases and new test cases.

C. Study result report and discussion

Since we have many testing groups to work on the case study, we selected some good study results from four groups to report the test case reuse and bug checking. Those groups performed the experiments strictly using our approach. To perform a complete regression testing process, we classified the regression test cases into newly created test cases, reusable test cases and deleted test cases. Those groups also reported the bugs found in regression testing.

![Figure 6. Regression Testing Result](image)

Table I

<table>
<thead>
<tr>
<th>Level</th>
<th>Test case design</th>
<th>Test case executed</th>
<th>Test coverage</th>
<th>Bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Car</td>
<td>79</td>
<td>79</td>
<td>100%</td>
<td>5</td>
</tr>
<tr>
<td>Floor Panel</td>
<td>59</td>
<td>59</td>
<td>100%</td>
<td>4</td>
</tr>
<tr>
<td>System</td>
<td>159</td>
<td>159</td>
<td>100%</td>
<td>11</td>
</tr>
</tbody>
</table>

Figure 6 shows the regression testing results. We have obtained test cases update results from component level, i.e. car component and floor panel component to system level.

In the figure, there are three subgraphs, which represents the study results of car component, floor panel component and system respectively. The horizontal axis represents new test cases, reused test cases and deleted test cases respectively. The data of test cases are represented by a vertical bar. The height of the bar depicts the number of new, reusable, and deleted test cases.

From the figure, we can find, after regression testing, some of the original test cases could be reusable and some could be deleted. In addition, new test cases need to be created for the new version system to achieve adequate testing. For instance, in Figure 6, 25 test cases for car component, 23 test cases for floor panel component and 29 test cases for system level are newly created. This is because we have added a new component indicator in the elevator system. The indicator is added to both car component and floor panel component.

We also find most of the test cases are reusable. For instance, 48 test cases for car component, 44 test cases for floor panel component and 60 test cases for system level are reusable. The explanation is that most of the components in the system are reused in the new version. Several test cases are obsolete and deleted after regression testing due to the program changes.

Table 1 represents the bug report. The test case design includes all the new test cases and reused test cases. All of those test cases are executed with a 100% coverage. The bugs come from both component level and system level. For example, 4 bugs in floor panel component and 5 bugs in car component are found. In addition, 11 bugs are found in system. Thus, the modification does bring affection and impact on both the component and system.

From the result of case study, we can see the proposed approach can obtain reusable, deleted and new test cases at both levels. In addition, we also have bug reports. Hence, our approach is feasible when applied to real component-based software system.

In the case study, Most of the test cases are reusable. In terms
of the statistics of studies, nearly over 70% of the test cases are reusable. That means most of our original test cases are kept for further testing. Thus, we still can achieve a relatively high test coverage. Some test cases are deleted after regression testing, which meets the demand of reducing the number of obsolete original test cases for cost-effectiveness. Moreover, in the case study, several program bugs are reported by many groups. We also found the bugs do existed in the system after modifications. This indicates that our approach is effective.

VII. Conclusions and future work

This paper has presented a systematic regression testing technique for component-based software from component level to system level. We analyzed the whole process of regression testing, including change identification, change impact analysis and test case updating.

We proposed several re-models to support regression testing. The diverse types of changes are considered in this paper, such as component level API changes, interaction changes, and system level architecture changes. For impact analysis, the firewall concept is borrowed and extended to analyze affected program parts according to the proposed re-test models. For test models, we provide a commonly used component testing method-decision table-based testing. The change types are mapped to impact analysis, and then the affected parts are mapped to the affected test cases which are decision table-based. In addition, we performed case studies on a realistic component-based software system. The studies results show that our approach is feasible and effective.

The future extension of this research is to apply the approach into different component testing methods and models, such as state-based testing, scenario-based testing, etc. In addition, we will study how to use the approach to address automation regression test issues and develop automatic component-based regression testing tools.

References