A Systematic State-Based Approach to Regression Testing of Component Software

Chuanqi Tao[1,2,3], Bixin Li[1,2]*, Jerry Gao[3]
[1]School of Computer Science and Engineering, Southeast University, Nanjing, Jiangsu, China
[2] Key Laboratory of Computer Network and Information Integration (Southeast University), Ministry of Education
[3]Department of Computer Engineering, San Jose State University, San Jose, CA, USA

Abstract— **Today, component-based software engineering has been widely used in software construction to reduce project cost and speed up software development cycle. Due to software changes in new release or update of components, regression testing is needed to assure system quality. When changes made to a component, the component could be affected, moreover, the changes could bring impacts on the entire system. We firstly identify diverse changes made to components and system based on models, then perform change impact analysis, and finally refresh regression test suite using a state-based testing practice. Related existing research did not address the issue of systematic regression testing of component-based software, especially at system level. The paper also reports a case study based on a realistic component-based software system using a, which shows that the approach is feasible and effective.**

Index Terms— **component-based software regression testing; re-test model; state-based testing; change and impact analysis; test suite refreshment**

I. Introduction

A component-based software system is primarily constructed based on reusable components, such as third-party and in-house built components. When a component is updated or upgraded, it must be re-tested. This refers to regression testing. 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 a component-based software, Changes made to a component could cause impact on the other parts of the component, other components of the system, and the entire system behaviors. Thus, regression testing should be conducted from component-level to system level. In addition, practical test models should be taken into account. For instance, if a state-based method has been chosen as a test model to generate test cases for original system version, then we should consider how to refresh test suites based on the state-based test suite in the updated version.

Research in the past addressed some component-based regression testing issues, such as UML-based [2], Metedata and glue code-based [3]–[5], API-based [6]. However, they did not address the issue of mapping changes and impacts into the affected test cases generated by specific testing. For example, if a component-based system has some changes made to API and its original testing is based on state. In this case, we need to analyze caused impacts due to API changes and refresh test suites based on state-based testing. The existing papers did not 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. Our previous work [7] briefly discussed the regression testing of component software using state testing practice, which is an initial work for conference. In this paper, we discuss the retest models, change impact analysis and test refreshment in detail. This paper addresses those needs above by providing a systematic approach to regression testing of component-based software based on re-test models, which are used to present the dependency relationships amongst components, assist engineers to define re-test criteria and re-integration strategies, and facilitate automatic test generation. This paper proposes several re-test 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.

Firewall is a well-known approach to change impact analysis [8], [9] introduced by Leung and White. Changes and affected parts are included in firewall, based on various dependencies, such as module dependencies, control-flow dependencies, etc. The firewall concept [8] is borrowed and extended 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. To identify the affected test cases, the software change firewall should be mapped into the corresponding test cases. Since the project of our case study is a control driven system and tested using state-based testing method, therefore, we regard a state chart as a basic test model for our regression testing in this paper. The related test suite refreshment is based on the state chart. The process to perform regression testing of component-based software from component to system is

This work is 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.

*Correspondence to: bx.li@seu.edu.cn

as follows:

☐ Change Impact analysis at component level, which refers to component function firewall, component data function firewall, and component API function firewall.

☐ Test suite refreshment at component level, which refers to component state-based test cases reused, changed, deleted and added.

☐ Change impact analysis at system level, which refers to component interaction firewall.

☐ Test suite refreshment at system level, which refers to system state-based test cases reused, changed, deleted and added.

The major contributions of this paper are summarized below:

(1) A systematic model-based solution to regression testing of component-based software from component to system level is presented, including change identification, change impact analysis and test suite refreshment.

(2) Several new component firewalls are introduced for software change impact analysis.

(3) A practical regression testing based on state-based testing is performed in our case studies.

This paper is organized as follows. Section II introduces a brief review of the related work in regression testing of component-based software. In section III, the re-test models are presented. Section IV analyzes the change types and provides a systematic approach to component-based software change impact analysis using firewall. Section V discusses test suite refreshment. Section VI 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 [10]–[14], regression testing cost-effectiveness analysis [15], [16], and object-oriented re-integration [8], [17]–[19].

A lot of published papers focused on the component-based software testing issues [20]–[22]. Currently, component-based software testing mainly focuses on component testability, component test adequacy and coverage, component-based software integration, performance testing, and configuration testing. In addition, some research focuses on component reuse [23] and reconfiguration [24], [25]. In the past years, a few papers addressed the regression testing problems existed in component-based software [2]–[6], [26]. They can be generally classified into the following three groups.

The first group focused on regression test selection of component-based software. For example, Harrold et al. proposed an approach to regression testing of COTS components using component meta-data [26]. 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 [3]. 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 [4]. 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 document should be visible. They assume that when components are changed 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 changed component. 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 [27].

The second group focused on UML-based. For instance, Wu et al. presented a UML technique for regression testing of component-based software [2]. In software maintenance activities, the technique 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. However, their state chart is based on UML and re-test analysis is directly based on statechart. In addition, regression testing of component-based software at system level was not taken into account.

The third group focused on 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 [6], [20]. In addition, Mao et al. proposed an improved regression testing method based on built-in test design for component-based system [28].

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

-What are the re-test models, strategies for component-based system due to component changes?

-How to identify change impact analysis from component to system due to specific component changes?

-How to refresh test suite based on change impact analysis? For example, how to refresh test cases from specific testing method, such as state-based or decision table-based testing?

This paper addresses those problems above. Unlike our previous work that only focused on component change analysis and impact at the component level [6], this paper proposes a model-based approach for regression testing

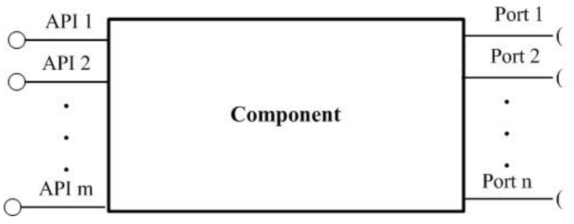


Figure 1. Component API model

according to a state-based testing practice. Moreover, this paper introduces a systematic methodology for regression testing of component-based software from component level to system level, including component-based re-test models, change impact analysis and test suite refreshment.

III. Component-based Software Re-test Models

In component-based software, 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. At component level, component changes could be API 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 such as *data function dependency graph* and *function dependency graph* could be adopted for component re-test [6]. Since components are usually called through API functions, API models can be borrowed for component re-test. From the black-box view, the component could be tested using *state-based* testing or *decision table-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. At system level, the relationship between components could be *interaction*. Thus, related system-level models can be used for system regression testing. For example, from the integration view, *component interaction graph* can be used as interaction models for system. From the *function* view, *state-based* methods could also be adopted for system function testing. As component system is usually configurable, the *composition* and *configuration* models are needed to support the system testing from the architecture view. However, architecture is out the scope of this paper.

In this paper, we conduct regression testing of component-based software from API view, interaction view, and state-based view. Next, we will introduce the associated re-test models in detail.

A. Component API Model

API model is the basic model in this paper. From the perspective of API, component-based software could be viewed as a specific black-box. For a given component, the interface of the black-box component includes API and port. API functions or data are the interface for users to access. Port represents the calling relationship between the component and the other components it calls. Figure 1 shows a sample component API model. In addition, component API functions have preconditions and postconditions, which can be obtained from component specification.

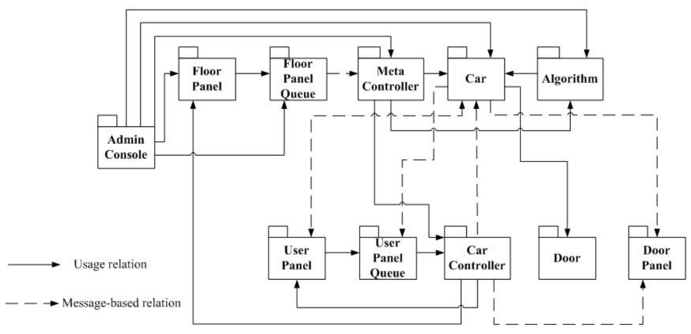


Figure 2. A Sample Component Interaction Graph of Partial Elevator System

B. Component-level Re-test Models

Existing white-box and black-box re-test models for conventional programs could be used to support component regression testing. At component-level, we primarily utilize two re-test models proposed in [6]: function dependency graph (FDG) and data function dependency graph (DFDG). FDG depicts function dependency through function call inside the component, and DFDG represents function dependency through data define-use relation. We do not discuss those two models in detail here.

C. System-level Re-test Model-Component Interaction Graph

Wu et al. introduced a *component interaction graph* to describe the interactions and dependency among components, which is mainly call relation [29]. In this paper, *Component Interaction Graph (CIG)* addresses the relations such as *message-communication* and *usage* for component-based software through API call. The extended component interaction graph is defined below.

Definition 1 *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 relation labels, and $E = E_{MSG} \bigcup E_{USA}$ 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 components. $E_{MSG} \subseteq N \times N \times R$ is the set of directed edges representing the message-based interaction relation between the components.*

For instance, For any two components $C_1, C_2 \in N$, $\langle C_1, C_2, MSG\rangle \in E_{MSG}$ indicates that component C_1 sends message to C_2. A sample CIG is shown in Figure 2. In the elevator system, *userpanel* uses *userpanelQueue*, so there is a usage relation between them. Besides, car needs

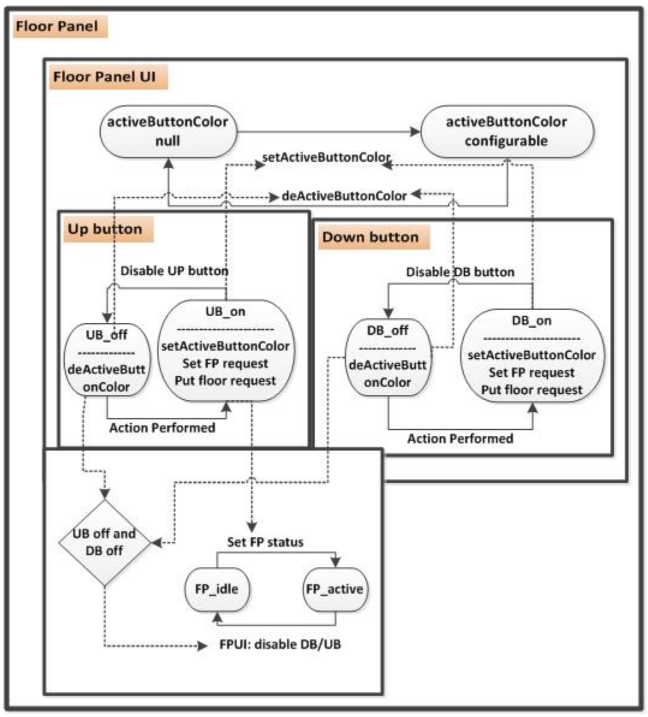


Figure 3. A Sample State Chart of Partial Elevator System

to communicate with userpanel to ensure the elevator system work normally, thereby they have a message communication relation.

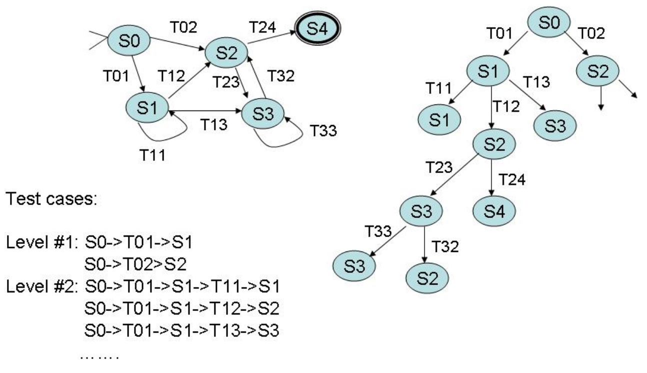
D. State Chart Model at Component Level

State-based testing is a commonly used component testing method. In this paper, regression testing is performed based on a state-based model. At component level, state chart represents the *state-transition* relations inside the component, which is similar to the traditional state chart. Here, *component state chart* can be defined below.

Definition 2 *The Component State Chart (CSC) for component-based software is a directed graph $CSC = (N_S, E, R)$, where N_S is the set of nodes representing component states. $R = SS$ is the action from state to state.* E *is the set of edges between nodes. $E \subseteq N_S \times N_S \times R$ is the set of directed edges representing the transition relation between the component states.*

E. State Chart Model at System Level

In component-based software, there exists interactions between components as we defined above. Similarly, if we model the system using a state chart, there might exist communication relations between states or transitions. Figure 3 presents a sample *state chart* of *floor panel* component in a elevator system (The system is used as our case study subject). This is a complex component, which means there exists communication relation between subcomponents. The solid line represents *transition* inside component and the dotted line represents *transition* between subcomponents. For instance, when Up-button (UB) is on, it will trigger the action *setactivebuttoncolor* in *Floor Panel UI*. Hence, UB-on can be considered as a



Test cases:

Level #1: S0->T01->S1
S0->T02>S2
Level #2: S0->T01->S1->T11->S1
S0->T01->S1->T12->S2
S0->T01->S1->T13->S3
…….

Figure 4. A Sample Test Tree Model

communication state (CS), which could trigger a series of actions to communicate with actions in other components, and the trigged action can be called *communication action* (CA).

Thus, we propose a *state chart* at system level, which takes into account the transitions from communication state to action. The *system state chart* is defined as follows.

Definition 3 *System State Chart (SSC) for component-based software at system level is a directed graph $SSC = (N_S \cup N_{CS} \cup N_{CA}, E, R)$, where N_S is the set of nodes representing component states, N_{CS} is the set of nodes representing communication state, and N_{CA} is the set of nodes representing communication action. R represents the set of actions. E is the set of edges representing transition relation. $E = E_{SS} \bigcup E_{SA}$. $E_{SS} \subseteq N_S \times N_S \times R$ is the set of directed edges representing the transition relation between the component states. For N_{CS} in component C_1 and N_{CA} in component C_2, $E_{SA} \subseteq N_{CS} \times N_{CA}$ is the set of directed edges representing a communication relation between component C_1 and C_2.*

F. Test Tree Model

To support the generation of state-based test cases, we introduce a test tree model. A *Test Tree* (*TT*) is a kind of *traverse tree* of the state chart. Breadth or depth first algorithms can be adopted to generate test tree. For the system state chart, the corresponding test tree can be considered as a test tree forest. The test tree model can be defined as follows:

Definition 4 *TT can be defined as a 3-tuple $G = (N_{TT}, E_{TT}, R_{TT})$, where*

- *N_{TT} is the set of tree nodes, including all the state nodes in the state chart. $N_{TT} = N_S \cup N_{CS}$. S_0, which is the start node of state chart, becomes the root node of TT.*
- *E_{TT} is the set of transition links between tree nodes. $E_{TT} = E_{SS} \cup E_{SA}$*
- *R_{TT} is the set of actions between states.*
- *For self-transition states, they are only traversed once in TT. For instance, in Figure 5, state S_1 and transition T_{11} in state chart is transformed into $S_1 \longrightarrow T_{11} \longrightarrow S_1$ in test tree model.*

Once the Test Tree is generated, we can obtain the test sequence at different tree levels incrementally.

$TT = \{L_1, L_2, ..., L_n\}$

$L_1 = \bigcup \langle S_0, N_{TT}(1) \rangle$

where $N_{TT}(1)$ is the tree nodes at level 1, which means the distance between the current node and root node is 1 and L_1 is the test case sequence set at level 1. For example, in Figure 4, the test cases at level 1 can be generated from following state-transition path:

TABLE I.
COMPONENT CHANGE TYPE

Type	Specific Changes
AID	Add an internal data
AIF	Add an internal function
AAF	Add an API function
APF	Add a Port function
DID	Delete an internal data
DIF	Delete an internal function
DAF	Delete an API function
DPF	Delete a Port function
CID	Change an internal data
CIF	Change an internal function
CAF	Change an API function
CPF	Change a Port function

$S0 \longrightarrow T01 \longrightarrow S1$
$S0 \longrightarrow T02 \longrightarrow S2$

Similarly,
$L_2 = \bigcup \langle S_0, N_{TT}(2) \rangle$

where $N_{TT}(2)$ is the tree nodes at level 2 and L_2 is the test case sequence set at level 2.

$$\vdots$$

$L_n = \bigcup \langle S_0, N_{TT}(n) \rangle$

where $N_{TT}(n)$ is the tree nodes at level n and L_n is the test case sequence set at level n.

IV. CHANGE IMPACT ANALYSIS

Change impact analysis is an important stage of component-based software regression testing. Some of the existing research assumed that the test model change information can be obtained from specification directly [21], [29]. However, sometimes the test model change information is not available. In this case, we need to perform change impact analysis firstly and then match the related change and impact information with the affected parts in the updated test model, so as to identify the affected test cases, and refresh the test suites.

Different change types might result in diverse impact. We have summarized the common changes existed in component-based software which is shown in Table I. Then, we analyze various change impacts based on the specific change types. To identify component change impacts at both levels, some detailed algorithms and methods are needed to support the solution. Change impact analysis is based on the re-test models proposed in Section III. We introduce several component change firewalls, which include *change firewall*, *add firewall* and *delete firewall*. In our approach, the basic procedure to perform change impact analysis consists of the following steps:

- Identify the impacts of a changed component function or data on other component functions, then we choose the affected APIs, and identify the component change impacts on its API, and its precondition, action, and postcondition at component level.

- Identify the component change impact on other components and their APIs or ports based on component interaction relation at system level.

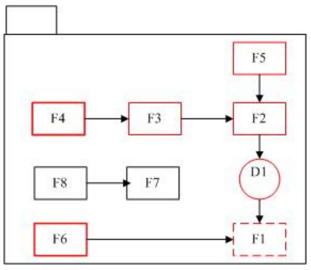


Figure 5. A Sample Change Firewall at Component Level

A. *Component Function Dependency Firewall*

A small change of any component in a system could cause ripple effects on other components and system behaviors. In previous work, we proposed some firewalls, such as *component function firewall*(CFFW) and *component data dependency firewall* (CDFW)at component level. Through the computation of *CFFW* and *CDFW*, we can get the affected component functions based on invocation dependencies and data *define-use* dependencies [6]. Various change types correspond to different impact. Now we try to present the change impact analysis corresponding to the summarized change types in Table 1 using firewall. Based on the proposed retest models, we compute the firewall using graph reach-ability theory, i.e., the nodes which could reach the changed node are identified through various relations in the models. Here, the changed data is assumed as D_i, and the changed function is assumed as F_i. We have the following change impact computation formulas.

- For adding a data (change type: AID)
  $CDFW_{add}[D_i] = \{F_j | (\exists F_j)(\exists F_k)(((\langle F_k, F_i, du \rangle \in R'_d - R_d) \wedge (F_k \in F) \wedge (\langle F_j, F_k, du \rangle \in R'_d*))\}$
  Where R'_d is the data define-use relation derived from DFDG', and R'_d* represents the transition closure relation for R'_d.
- For adding a function (change type: AIF, AAF, or APF)
  $CFFW_{add}[F_i] = \{F_j | (\exists F_j)(\exists F_k)(((\langle F_k, F_i \rangle \in E' - E) \wedge (F_k \in F) \wedge (\langle F_j, F_k \rangle \in R'_f*))\}$
  Where R'_f is the dependence relation for FDG'.
- For deleting a data (change type: DID)
  $CDFW_{delete}[D_i] = \{F_j | (\exists F_j)(\exists F_k)(((\langle F_k, F_i, du \rangle \in R_d - R'_d) \wedge (F_k \in F) \wedge (\langle F_j, F_k, du \rangle \in R'_d*))\}$
- For deleting a function (change type: DIF, DAF, or DPF)
  $CFFW_{delete}[F_i] = \{F_j | (\exists F_j)(\exists F_k)(((\langle F_k, F_i \rangle \in E - E') \wedge (F_k \in F) \wedge (\langle F_j, F_k \rangle \in R'_f*))\}$
  Where R'_f is the dependence relation for FDG'.
- For changing a data (change type: CID)
  $CDFW_{change}(D_i) = \{F_i | ((\langle F_i, D_j \rangle \in R_{def}) \wedge ((\langle F_j, D_j \rangle \in R_{use}) \wedge ((\langle F_j, F_i, du \rangle \in R*_{dr})\}$
  Where $R*_{dr}$ is the transition closure of R_{dr}, which is the binary relation that define the data define-use relation between the residual component functions. R_{dr} can be defined as:
  $R_{dr} = R_d \cap (F \times F \times \{du\}) \cap (F' \times F' \times \{du\})$
- For changing a function (change type: CIF, CAF, or CPF)
  $CFFW_{change}(F_i) = \{F_j | (F_j, F_i \in F) \wedge ((\langle F_j, F_i \rangle \in R*_{fr})\}$
  Where $R*_{fr}$ is the transition closure of R_{fr}, which is the binary relation that define the dependencies between the residual component functions. R_{fr} can be defined as:
  $R_{fr} = R_f \cap (F \times F \times \{du\}) \cap (F' \times F' \times \{du\})$, where x is the Cartesian product operation.

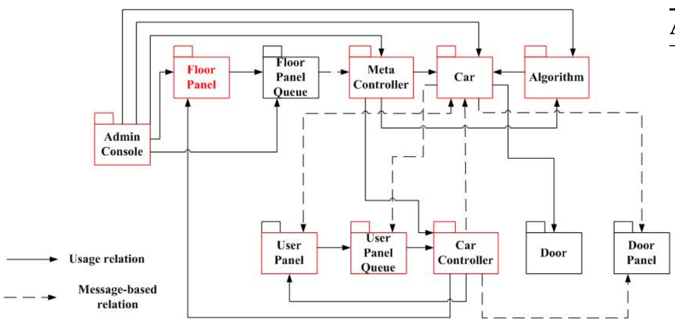


Figure 6. A Sample Component Interaction Graph Firewall in the Elevator System

Regarding API function firewall, our previous work already discussed it [6]. The definition is as follows.

$CAW_{API}(C) = \{F_i | \forall F_i((F_i \in F'_{API} - F_{API}) \wedge ((F_i \in CFFW(C)) \vee (F_i \in CDFW(C))))\}$

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.

Figure 5 shows a sample change firewall in component *function data dependence graph* (FDG) at component level. Assuming F1 is changed, according to the change firewall computation, the affected function and data in the firewall is marked red in the graph. In API change firewall, only the affected API functions are kept. Thus, API function firewall $CAW_{API}(C)$ incorporate F4 and F6 (assuming they are API functions).

From the component specification, we can obtain component APIs and their preconditions and postconditions. Now we extend $CAW_{API}(C)$ by adding the *precondition*, *postcondition*, 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.

$ECAW(C) = \{\langle prec_i, postc_j, action_k \rangle | (prec_i$ is changed, added or deleted) $\vee (postc_j$ is changed, added or deleted) $\vee (action_k \in CAW_{API}(C))\}$

Where $prec_i$, $postc_j$, $action_k$ denote *precondition*, *postcondition* and *action* respectively. $\langle prec_i, postc_j, action_k \rangle$ stands for a vector set, which include the affected *preconditions*, *postconditions* and *actions*.

As we mentioned before, the precondition, postcondition, and action of API can be transformed into transitions and states in state chart directly. Thus, we can easily match the change firewall with the affected state-based test cases, which will be discussed later.

B. Component Interaction Graph Firewall

A component function or data change not only affects itself, but also brings impacts on functions or data in other components. Thereby we need to analyze the change impacts based on interaction relation at system level.

Algorithm 1 Component Interaction Graph Firewall

```
Declare:
CIG: component interaction graph;
$CIG'$: modified component interaction graph;
$C_i.F_i$: changed function $F_i$ in component $C_i$;
$CIGF_{API}(C_i.F_i, CIG, CIG')$: component interaction graph firewall
$CIGF_{API}(C_i.F_i, CIG, CIG')$
{
switch (change type)
  case 'Add functions':
    $CFFW_{add}[C_i.F_i]$;
    $CDFW_{add}[C_i.F_i]$;
      $CFFW_{add}[C_i.F_i] \cup CDFW_{add}[C_i.F_i]$;    //compute affected functions inside component $C_i$
    $C.F = ECAW[C_i.F_i]$;    //compute affected API functions
    break;
  case 'delete functions':
    $CFFW_{delete}[C_i.F_i]$;
    $CDFW_{delete}[C_i.F_i]$;
    $CFFW_{delete}[C_i.F_i] \cup CDFW_{delete}[C_i.F_i]$;
    $C.F = ECAW[C_i.F_i]$;
    break;
  case 'change functions':
    $CFFW_{change}[C_i.F_i]$;
    $CDFW_{change}[C_i.F_i]$;
    $CFFW_{change}[C_i.F_i] \cup CDFW_{change}[C_i.F_i]$;
    $C.F = ECAW[C_i.F_i]$;
    break;
Mark each function in C.F visited;
put $C.F$ in $CIGF_{API}$;
$C_j = CIG[C_i].rlink$;    // $\langle C_j, C_i \rangle \in R_{CIG}$
While ( $C_i.f_i$ in $C.F$ ) do    // compute affected functions in other components
{                                   // based on interaction relation
if $(C_j.F_j, C_i.f_i) \in P(C_j) \wedge C_j.F_j$ not visited
then
$CIGF_{API}(C_j.F_j, CIG, CIG')$
}
}
```

Here, *Component Interaction Graph Firewall* (CIGF) refers to a set of component APIs which may be affected by changing, adding and deleting components or functions based on interaction dependencies. The *Component Interaction Graph Firewall* can be computed based on the given *Component Interaction Graph*. From the API view, CIGF can be utilized to include the affected API functions inside the firewall through interaction and invocation dependency. The *Component Interaction Graph Firewall* algorithm is shown in Algorithm 1. In the algorithm, for each change type, the API firewall inside component is computed firstly, then API function interaction between different components are considered. Finally, the affected API functions due to changes are incorporated in *Component Interaction Graph Firewall*. In the case of the change types in Table II, the firewall could be computed by Algorithm 1. A sample CIGF is shown in Figure 6, where *floorpanel* component is changed, and the affected components via interaction dependencies are marked highlighted with red color.

V. Test Suite Refreshment

The final step of regression testing is to refresh the existing test cases, which includes selecting reusable test cases, changing test cases, deleting out-of-date test cases, and adding new test cases.

As we mentioned above, state-based testing is the major method to generate test cases in our approach. Now we need to analyze how to map the change impact firewall into affected test cases in the given test suite. In general, a state chart consists of a series of states and transitions, where transitions can be deemed component functions and state can be represented by preconditions and postconditions. That indicates component API function and its conditions associate with the state and its transitions. Therefore, we can identify the affected transitions from API function related firewall, and identify the affected states from preconditions and postconditions of API function. Finally, we can refresh test suite according to the updated state chart and its test tree.

The procedure to perform test suite refreshment can be divided into two steps: 1) identify the change impacts on component state-based test cases and refresh test suite at component level; 2) identify the change impacts on system state-based test cases and refresh test suite at system level.

A. *Test Suite Refreshment at Component Level*

Let's assume a state-based test cases CTS_i is generated based on the *test tree* which is defined in section III, and executes a state-transition path P_i of *test tree*. Through the computation of firewall $ECAW$, the affected actions, precondition, and postcondition can be identified (Section IV). Then, we can get the affected states and transitions in corresponding state chart. According to the affected test tree, we can obtain affected test cases. Assuming TT is the original test tree and TT' is the modified test tree, and P_i is the original state-transition path and P_i' is the modified state-transition path for updated component version, we have the following analysis of test refreshment.

(1) A reusable state-based test cases CTS_{reuse} can be identified as follows: For any CTS_i in CTS, it is reusable when all of the following conditions hold:

- There exits P_i and P_i' in TT and TT' respectively.
- All nodes and links of P_i and P_i' are the same, and any node and link of P_i' doesn't belong to any add, delete, and change firwall.

According to the analysis above, we can get the state-based test cases refreshment formally. For any test case CTS_i in *state test tree* (N_{TT}, E_{TT}, R_{TT}), it belongs to reused component test suite set CTS_{reuse}, if it satisfies the formula as follows:

$CTS_{reuse} = \{CTS_i | (\forall(N_{TT} \in (P_i \cap P_i'))(N_{TT} \notin ECAW)) \wedge (\forall(E_{TT} \in (P_i \cap P_i'))(E_{TT} \notin ECAW))\}$

Where ECAW includes add, delete and change firewall. The explanation to the formula is that for a given state-transition path existed in both original and updated test tree model, if each node or edge in the path is not affected, i.e., not included in firewall, then the corresponding test cases could be reused.

(2) An out-of-date state-based test case CTS_{delete} could be identified as follows:

For any CTS_i in CTS, it should be deleted when any of the following conditions hold:

- Any single node in P_i is deleted from TT and does not exist in TT'.
- Any single link in P_i is deleted from TT and does not exist in TT'.

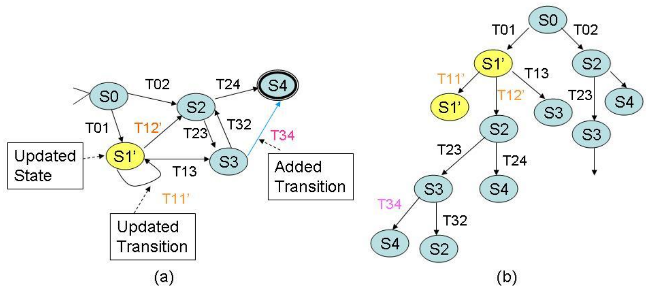


Figure 7. A Sample Updated State Chart and the Corresponding Test Tree

Thus the set for deleted component test cases can be formally computed below.

$CTS_{delete} = \{CTS_i | (\exists(N_{TT} \in P_i)(N_{TT} \in (TT - TT'))) \vee (\exists(E_{TT} \in P_i)(E_{TT} \in (TT - TT')))\}$

(3) A updated state-based test case CTS_{change} can be identified as follows:

For any CTS_i in CTS, it should be revisited and updated when all of the following conditions hold:

- P_i exists in both TT and TT'.
- There is at least one node or one link in P_i has been changed or affected (in the change firewall).

Thereby the set for changed component test cases can be formally identified below.

$CTS_{change} = \{CTS_i | (\exists(N_{TT} \in (P_i \cap P_i'))(N_{TT} \in ECAW)) \wedge (\exists(E_{TT} \in (P_i \cap P_i'))(E_{TT} \in ECAW))\}$

(4) A new state-based test case CTS_{new} for the updated component could be generated to cover any new path in TT'. The set for new component test cases can be formally computed below.

$CTS_{new} = \{CTS_i | (\exists(N_{TT} \in P_i)(N_{TT} \in (TT' - TT))) \vee (\exists(E_{TT} \in P_i)(E_{TT} \in (TT' - TT)))\}$

For instance, in Figure 7(a), there exists some changes made to the state chart which are caused by component or system changes and impacts. Here we have three changes in the state chart as follows: (a) state S1' is updated; (b) transition T11' is updated; (c) T34 is a added transition. According to the test tree model, the corresponding test tree for the updated state chart is shown in Figure 7(b). Therefore, the test suite refreshment could be summarized below:

Level 1:

$S0- > T01- > S1(updated)$
$S0- > T02 > S2(reusable)$

Level 2:

$S0- > T01- > S1- > T11- > S1(updated)$
$S0- > T01- > S1- > T12- > S2(updated)$
$S0- > T01- > S1- > T13- > S3(updated)$
......
$S0- > T02- > S2- > T24- > S4(reusable)$

B. *Test Suite Refreshment at System Level*

System state chart depicts the states and the transitions of complex component or the whole system. Similar to state-based test case firewall at component level, the state can be represented by preconditions and postconditions and transitions can be deemed system functions. Through the computation of $CIGF_{API}$, we can get the API function firewall at system level. Then, the affected preconditions, postconditions and actions could be mapped into the corresponding states and transitions. Hence, we can obtain the test suite refreshment with added, changed, deleted, and reused test cases. Here, STS_{reuse}, STS_{delete}, STS_{change}, STS_{new} represents the set for identified reused, deleted, changed, and new system test cases respectively. Assuming TT is the original test tree and TT' is the modified test tree, and P_i is the original state-transition path and P_i' is the modified state-transition path for updated component version, for any test case STS_i in *state test tree* (N_{TT}, E_{TT}, R_{TT}), the formulas are given below.

$STS_{reuse} = \{STS_i | (\forall (N_{TT} \in (P_i \cap P_i'))(N_{TT} \notin CIGF_{API})) \wedge (\forall (E_{TT} \in (P_i \cap P_i'))(E_{TT} \notin CIGF_{API}))\}$

Where STS_{reuse} represents the reused state-based test cases at system level. $CIGF_{API}$ is the API firewall calculated via Algorithm 1.

Similarly, we have the computation formulas for deleted, changed, and new test suite below.

$STS_{delete} = \{STS_i | (\exists (N_{TT} \in P_i)(N_{TT} \in (TT - TT'))) \vee (\exists (E_{TT} \in P_i)(E_{TT} \in (TT - TT')))\}$

$STS_{change} = \{STS_i | (\exists (N_{TT} \in (P_i \cap P_i'))(N_{TT} \in CIGF_{API})) \wedge (\exists (E_{TT} \in (P_i \cap P_i'))(E_{TT} \in CIGF_{API}))\}$

$STS_{new} = \{STS_i | (\exists (N_{TT} \in P_i)(N_{TT} \in (TT' - TT))) \vee (\exists (E_{TT} \in P_i)(E_{TT} \in (TT' - TT)))\}$

VI. Empirical Study

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 simulation system. The subject of the case study is a component-based configurable elevator simulation system developed by our students. The system (version 1) is well developed according to the component design principles. The elevator system consists of several components, which are car, user panel, door, door panel, userpanel queue, car controller, floor panel and metacontroller. The system is well designed with adequate component test cases and system test cases based on state testing. The students are trained to utilize the proposed approach in this paper. They performed change identification, impact analysis, and retesting using the proposed regression testing approach. The corresponding re-test models are created manually, based on dependency information. Then, the fiewalls such as CIGF are computed using the proposed firewall formulas and algorithms. Finally, according to the proposed test case update rules, the firewalls are mapped into affected test cases to determine the added test cases, reused test cases, deleted test cases, and changed test cases.

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.

TABLE II.
The Summarized Elevator System

Component Name	No. of Classes	No. of Functions	Size (Loc)	No. of Source Code Files
AdminConsole	7	32	912	4
Algorithm	6	7	190	6
Car	12	86	579	11
CarController	4	8	111	4
Door	9	35	309	7
DoorPanel	7	36	224	8
FloorPanel	7	28	376	7
FloorPanelQueue	5	16	197	4
MetaController	5	13	970	5
UserPanel	11	61	647	11
UserPanelQueue	8	26	238	8

B. *Study Subject*

The study subject is a component-based elevator simulation system. The system consists of several components, which are *car*, *user panel*, *door*, *door panel*, *userpanel queue*, *car controller*, *floor panel* and *metacontroller*. The summary of the component-based elevator system is presented in Table II.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 state-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, such as FCFS and SCAN. In the new version, we conducted regression testing from component level to system level using the proposed approach. The user interface for the original elevator system is shown in Figure 8. The user interface for one version of modified elevator system with adding *indicator* is shown in Figure 9.

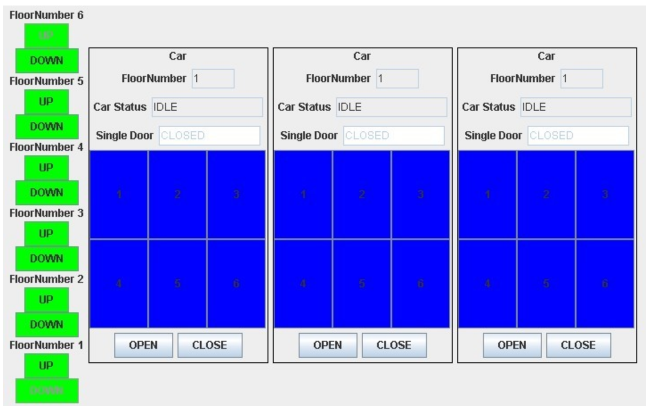


Figure 8. The User Interface for the Original Elevator System

C. *Study description*

The students are trained to utilize the proposed approach in this paper. They performed change identification, impact analysis, and retesting using the proposed regression testing approach. The corresponding re-

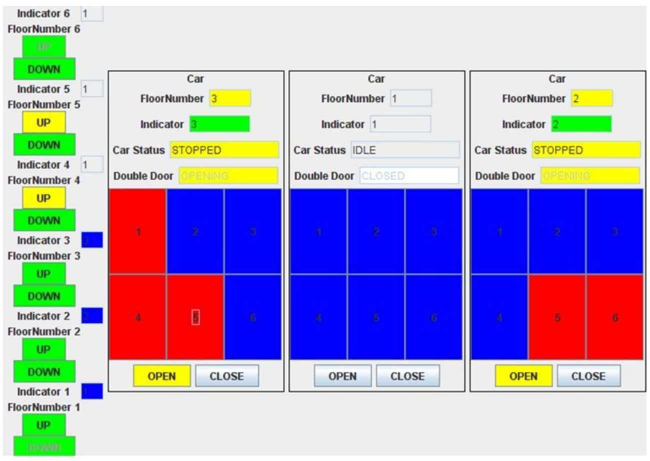


Figure 9. The User Interface for one version of Changed Elevator System

test models are created manually, based on dependency information. Then, the firewalls such as CIGF, ECTF are computed using the proposed firewall formulas and algorithms. Finally, according to the proposed test case update rules, the firewalls are mapped into affected test cases to determine the added test cases, reused test cases, deleted test cases, and changed test cases.

D. *Study result report and discussion*

We chose two project groups to report the study results. Table III presents the recorded change information at both component level and system level from those two groups. According to summarized change types, the number of associated changes including internal data and function, interactions such as message and port, and architectures such as composition and configuration are listed in the table. At component level, Group 1 added 21 internal functions 10 internal data, and Group 2 added 12 internal functions and 5 data respectively. Here, adding is the major changes, since our change requirement is adding features. Regarding interaction changes, Group 1 added 8 APIs and 14 messages, and Group 2 added 7 APIs, changed 4 APIs and added 1 port. At architecture system level, Group 1 added 4 compositions and 3 configurations, which Group 2 only added 1 composition and 1 configuration. In summary, the changes involve both component level and system level, and Group 1 made more changes than Group 2.

Table IV shows the impact information at component level and systemlevel. The impact type represents the affected component elements and relationships which are computed via firewall analysis. Here we list the number of affected impact types for component, interaction, and architecture respectively. In general, those two groups have the similar impacts. They both have 11 affected internal data and functions and 1 affected configuration. In addition, they have 4 and 5 affected APIs, 3 and 4 affected messages, 1 and 2 affected compositions respectively. However, Group 1 has 4 affected ports, which are not reported by Group 2. Table V records the affected preconditions, actions, and postconditions of API, i.e., the update information of the new state chart and test tree for the elevator system Version 2. Note that *add* represents the new conditions or actions are added, *delete* represents the obsolete conditions or actions are deleted, and *change* represents affected conditions or actions which are collected through firewall computation.

Figure 10 and Figure 11 show the results of test case update from Group 1 and Group 2 respectively. The two subgraphs represent the component and system test case update information respectively. The horizontal axis represents added, reused, deleted and changed test cases respectively, and the height of the bar depicts the number of those test cases.

From the figure, we find 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 10, from the result of Group 1, 45 component test cases and 37 system test cases are newly created. In Figure 11 from Group 2, 21 component test cases and 29 system test cases are newly created. We also find most of the test cases are reusable. In Group 1, 92 component test cases and 105 system test cases are reusable. In Group 2, 117 component test cases and 130 system test cases are reusable. The explanation is that most of the components in the system are reused in the new version. In addition, Several test cases are changed and deleted after regression testing due to the program changes.

To investigate the effectiveness of our approach, we need a comparison criterion. In traditional regression test selection, there exists some criteria such as *precision*, *inclusiveness*, *execution time*, and *number of test cases* [30], [31]. Since our approach firstly proposes the strategy for regression testing of component-based software from component to system in existing work, we adopt *number of test cases* to compare with re-test all strategy, which selects all the original test cases. Here, we utilize the percentage of selected test cases out of the total original test cases to indicate the effectiveness of the approach. Table VI shows the percentage of selected test cases by our approach and related bug report. Group 1 and Group 2 selected 64.3% and 81.8% of the component test cases, and 61.8% and 76.5% of the system test cases respectively. That indicates they both obtain significant test cases reduction compared to re-test all strategy, which selected 100% of the original test cases. Additionally, we have the bug reports from Group 1 and Group 2 respectively. For instance, 9 bugs at component level are reported by both Group 1 and Group 2, and 11 and 13 bugs at system level are reported by them respectively. Moreover, all of these reported bugs are actual bugs, therefore, our approach is effective to reduce redundant test cases and detect bugs in practice.

From the result of case study, we can see the proposed approach can obtain added, reused, deleted, and changed test cases at both levels. In addition, we also have bug reports. Hence, our approach is feasible and effective

TABLE III.
COMPONENT AND SYSTEM CHANGE RECORD

Change Level	Change Type	Group 1			Group 2		
		Add	Delete	Change	Add	Delete	Change
Component Level	Internal Data	21	1	4	12	-	3
	Internal Function	10	-	4	5	-	1
System Level Interaction	API	8	-	-	7	-	4
	Message	14	-	-	-	-	-
	Port	-	-	-	1	-	-

TABLE IV.
COMPONENT AND SYSTEM IMPACT RECORD

Impact Level	Impact Type	Group 1	Group 2
Component Level	Affected Internal Data	8	3
	Affected Internal Function	3	8
System Level Interaction	Affected API	4	5
	Affected Message	3	4
	Affected Port	4	-

TABLE V.
COMPONENT AND SYSTEM CONDITIONS AND ACTIONS UPDATE RECORD

Update Level	Update Factor	Group 1			Group 2		
		Add	Delete	Change	Add	Delete	Change
Component Level	Precondition	16	4	6	7	3	5
	Action	21	13	5	9	8	7
	Postcondition	8	18	9	4	6	4
System Level	Precondition	12	8	5	10	5	3
	Action	15	19	16	14	12	8
	Postcondition	13	11	9	6	5	8

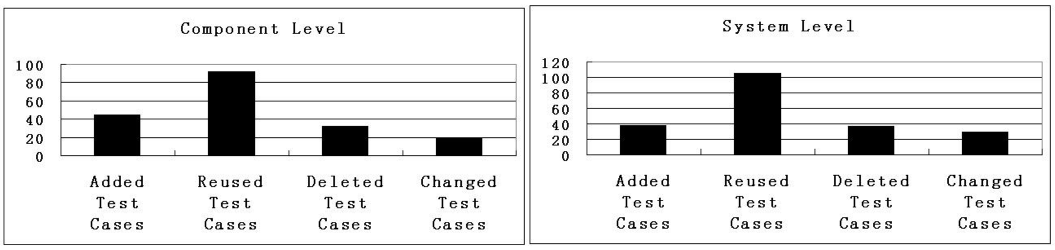


Figure 10. The Result of Test Suite Refreshment from Group 1

TABLE VI.
REGRESSION TESTING RESULT METRICS AND BUG REPORT

Metrics	Group 1	Group 2
Selected Component Test cases	64.3%	81.8%
Selected System Test cases	61.8%	76.5%
Reported Bugs at Component Level	9	9
Actual Bugs at Component Level	9	9
Reported Bugs at System Level	11	13
Actual Bugs at System Level	11	13

when applied to real component-based software system.

E. *Threats to validation*

There are several potential threats to the empirical study. Firewall-based approach can not guarantee all the impacts for regression testing are included in the firewall, which means there might exist change impacts or program faults introduced by changes outside the firewall. In addition, we did not consider the GUI change and impact, external environment software and hardware changes. The test cases are function-based, especially at system level, thereby some program faults or bugs probably cannot be revealed. The component-based system in the case study is built for academic use, hence, the size of the system is relatively not large enough in complex industrial environment. Although the students are trained to conduct the related experiments, there might still exist mistakes in the empirical studies.

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 proposed several re-test models to support regression testing. The diverse types of changes are considered, and for change impact analysis, the firewall concept is borrowed and extended to analyze affected program parts according to the proposed re-test models. State-based testing is used as a practical testing method. The change types are mapped to impact analysis, and then the affected parts are mapped to the affected test

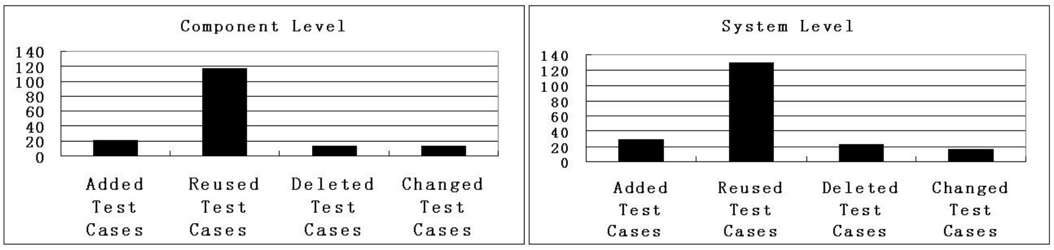


Figure 11. The Result of Test Suite Refreshment from Group 2

cases which are state-based. We performed case studies on a realistic component-based software system. The study results show that our approach is feasible and effective.

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

ACKNOWLEDGMENT

We thank Naftaly Minsky for numerous discussions concerning this work, Xuhui Ao and Constantine Serban for their assistance with LGI, and the reviewers for their detailed comments.

REFERENCES

[1] H. K. N. Leung and L. J. White, "Insights into regression testing," in *Proceedings of International Conference on Software Maintenance*, 1989, pp. 60–69.

[2] Y. W. et al., "Techinques of maintaining evolving component-based software," in *IEEE International Conference on Software Maintenance (ICSM 2000)*, 2000.

[3] A. O. et al., "Using component metacontents to support the regression testing of component-based software," in *Proceedings from the ICSE Workshop in Component-based software engineering*, 2001.

[4] J. Z. et al., "Applying regression test selection for cots-based applications," in *Proceedings of International Conference on Software Engineering*, 2006, pp. 512–522.

[5] J. Zheng, "In regression testing selection when source code is not available," in *Proceedings of International Conference on Automated Software Engineering*, 2005, pp. 752–755.

[6] J. G. et al., "A systematic regression testing method and tool for software components," in *Proceedings of the 30th Annual International Computer Software and Applications Conference*, 2006, pp. 455–456.

[7] C. Q. Tao, B. X. Li, and J. Gao, "Regression testing of component-based software: A systematic practise based on state testing," in *International High Assurance Systems Engineering Symposium*, 2011, pp. 29–32.

[8] L. White and H. K. N. Leung, "A firewall concept for both control-flow and data-flow in regression integration testing," in *Proceedings of the IEEE International Conference on Software Maintenance*, 1992, pp. 262–271.

[9] L. W. et al., "Extended firewall for regression testing: an experience report," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 20, no. 6, pp. 419–433, 2008.

[10] A. O. et al., "Scaling regression testing to large software systems," in *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2004, pp. 241–251.

[11] J. B. et al., "A comparative study of coarse- and fine-grained safe regression test-selection techniques," *ACM Transactions on Software Engineering and Methodology*, vol. 10, no. 2, pp. 149–183, 2001.

[12] G. Rothermel and M. J. Harrold, "Empirical studies of a safe regression test selection technique," *IEEE Transactions on Software Engieering*, vol. 24, no. 6, pp. 401–419, 1998.

[13] C. Q. Tao, B. X. Li, and X. B. Sun., "An approach to regression test selection based on hierarchical slicing technique," in *IEEE 23rd International Computer Software and Applications Conference (COMPSAC 2010)*, 2010.

[14] L. Briand, Y. Labiche, and S. He, "Automating regression test selection based on uml designs," *Information and Software Technology*, vol. 51, no. 1, pp. 16–30, 2009.

[15] G. R. et al., "The impact of test suite granularity on the cost-effectiveness of regression testing," in *Proceedings of International Conference on Software Engineering*, 2002, pp. 19–25.

[16] C. Q. Tao, B. X. Li, and X. B. Sun., "A hierarchical model for regression test selection and cost analysis of java programs," in *IEEE 2010 Asia Pacific Software Engineering Conference (APSEC2010)*, 2010, pp. 290–299.

[17] J. G. et al., "Identifying polymorphim change and impact in oo software maintenance," *Journal of Software Maintenance: Research and Practice*, vol. 8, no. 6, pp. 305–314, 1996.

[18] D. Kung and J. Gao, "On regression testing of object-oriented programs," *Journal of Systems and Software*, vol. 32, no. 1, pp. 21–40, 1996.

[19] ——, "Change impact identification in object-oriented software maintenance," in *Proceedings of the IEEE International Conference on Software Maintenance*, 1994, pp. 202–211.

[20] J. G. et al., "Testing and quality assurance for component software," *Massachusetts: Artech House, Inc*, 2003.

[21] Y. W. et al., "Uml-based integration testing for component-based software," in *The 2nd International Conference on COTS-based Software*, 2003, pp. 251–260.

[22] S. Beydeda and V. Gruhn, "Testing commercial-off-the-shelf components and systems," *Springer; 1 edition*, 2005.

[23] G. Gui and P. D. Scott, "Measuring software component reusability by coupling and cohesion metrics," *Journal of Computers, Special Issue: Selected Papers of ICYCS 2008*, vol. 4, no. 9, pp. 797–805, 2009.

[24] Q. Liu, B. Q. Wang, W. W. Huang, and N. Jiang, "Research on the reconfiguration router unit component composition technology based on the agent," *Journal of Computers, Special Issue: Recent Trends and Advances in Computer Science-Technology and Applications*, vol. 4, no. 9, pp. 218–225, 2010.

[25] W. Li, "Qos assurance for dynamic reconfiguration of

component-based software systems," *IEEE Transactions on Software Engieering*, vol. 38, no. 3, pp. 658–676, 2012.

[26] M. J. H. et al., "Using component metacontents to support the regression testing of component-based software," in *IEEE International Conference on Software Maintenance*, 2001, pp. 716–725.

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

[28] C. Y. Mao, "Regression testing for component-based software via built-in test design," in *ACM Symposium on Applied Computing*, 2007.

[29] Y. W. et al., "Techniques for testing component-based software," in *The 7th International Conference on Engineering of Complex Computer Systems*, 2001, pp. 222–232.

[30] G. Rothermel and M. J. Harrold, "Analyzing regression test selection techniques," *IEEE Transactions on Software Engieering*, vol. 22, no. 8, pp. 529–551, 1996.

[31] T. L. G. et al., "An empirical study of regression test selection techniques," *ACM Transactions on Software Engineering and Methodology*, vol. 10, no. 2, pp. 184–208, 2001.