Edge Coverage Analysis for Software Architecture Testing

Lijun Lun
College of Computer Science and Information Engineering, Harbin Normal University, Harbin, China
Email: lunlijun@yahoo.cn

Xin Chi
College of Computer Science and Information Engineering, Harbin Normal University, Harbin, China
Email: chixin9010@yahoo.cn

Xuemei Ding
Faculty of Software, Fujian Normal University, Fuzhou, China
Email: dxmgw@yahoo.com.cn

***Abstract*—Software architecture is perceived as one of the most important artifacts created during a system's design, to control software complexity, improve system quality, support software development and reuse and so on. Coverage analysis is a structural testing technique, which helps to eliminate gaps in a test suite and determines when to stop testing. To compute test coverage, the paper presents a new concept – coverage about edge based on C2-style architecture. Firstly, the software architecture is represented using C2-style, then we use architecture component interaction graph (CIG) to describe interface connection relationship, then we define three testing criteria and introduce algorithms to generate testing coverage set according to edge types of CIG. Finally, we present four edges coverage to compute coverage effectiveness.**

***Index Terms*—software architecture testing; C2-style; component interaction graph; edge coverage criteria; coverage analysis**

I. Introduction

Software architecture represents the earliest software design decisions. These design decisions are the most critical to get right and the most difficult to change downstream in the system development cycle. The software architecture is the first design artifact addressing reliability, modifiability, real-time performance, and inter-operability goals and requirements. Software architecture testing is an important technique for validating and checking the correctness of software architecture. Formalization testing based on software architecture has improved the quality of the software products. Automatic test coverage generation is a hotspot and difficulty in the field of software architecture testing. Current research divided into two categories [1]. One is to improve the traditional software testing techniques and methods, so that they service for software architecture testing. The other is to develop new software architecture testing techniques and methods, so that it can better solve problems of software architecture testing.

This paper uses C2-style architecture to model a software system, and apply the component interaction graph to software architecture testing. We have present methods to analyze test coverage for JAVA programs in our CASE tool.

The process can be divided into five steps: (1) Describe the software architecture using C2-style architecture, (2) Map C2-style specification to component interaction graph, (3) Analyze the dependence relationship between the interfaces and the events, (4) Define the three coverage criteria of component interaction and algorithms to generate testing coverage set, (5) Presents our approach to compute test coverage.

II. Basic Notions

This section introduces C2-style architecture formal defined, component interaction graph and its type of edge, and its built approach.

A. C2 Architecture Style

The C2-style architectural is primarily concerned with high-level system composition issues [2]. The C2-style architectural consists of components and connectors, which transmit messages between components. Components maintain state, perform operations, and exchange messages with other components via two interfaces (named top and bottom). Each interface consists of a set of messages that may be sent or received. Inter-component messages are classified into two types, viz. requests to a component to perform an operation, and notifications that a given component has performed an operation or changed state. In the C2-style architectural, both components and connectors have a top and a bottom interface. Systems are composed in a layered style, where a component's top interface may be connected to the bottom interface of a connector, and its bottom interface may be connected to the top interface of another connector. Each side of a connector may be connected to any number of components or connectors.

Definition 2.1 A C2-style architecture can be defined as C2 = (Comps, Conn), where:

- Comps = {Comps_1, Comps_2, …, Comps_m} is a finite set of components, where Comps_i = {Comps_i.top_in, Comps_i.top_out, Comps_i. bottom_in, Comps_i.bottom_out}.
- Conn = {Conn_1, Conn_2, …, Conn_n} is a finite set of connectors, where Conn_i = {Conn_i.top_in_1, Conn_i.top_in_2, …, Conn_i.top_in_n, Conn_i.top_out_1, Conn_i.top_out_2, …, Conn_i.top_out_n, Conn_i. bottom_in_1, Conn_i.bottom_in_2, …, Conn_i.bottom_ in_m, Conn_i.bottom_out_1, Conn_i. bottom_out_2, …, Conn_i.bottom_out_m}.
- bottom_in is the set of requests received at the bottom side of a component or connector. bottom_out is the set of notifications that a component or connector emits from its bottom side.
- top_in is the set of notifications received on the top side of a component or connector. top_out is the set of requests sent from its top side.

Fig. 1 represents the external view of a component Comps_i, Comps_i.top_in and Comps_i.top_out is defined by the component's dialog.

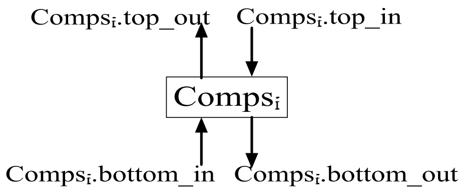


Figure 1. C2 component domains

Fig. 2 represents the external view of a connector $Conn_i$, with the components $Comps_{tj}$ (j = 1, ..., n) and $Comps_{ck}$ (k = 1, ..., m) attached to its top and bottom respectively. A connector's upper and lower domain is completely specified in terms of these components.

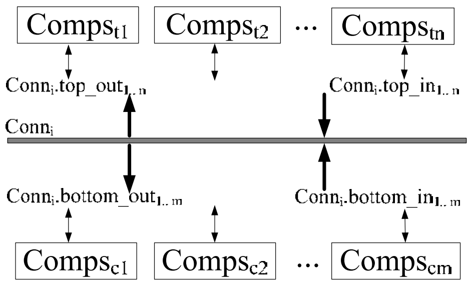


Figure 2. C2 connector domains

B. Component Interaction Graph

The component interaction graph is a digraph whose nodes represent the top or bottom of component or connector, and edges represent possible information flows between component and connector in the C2-ADL architecture specification.

Definition 2.2 Let C2 = (Comps, Conn) is C2-style architecture. Component interaction graph can be defined as direct graph CIG = (V, E, V_{start}, V_{end}), where:

- V = {Comps_i.top_in, Comps_i.top_out, Comps_i. bottom_in, Comps_i.bottom_out, Conn_j.top_in, Conn_j.top_out, Conn_j.bottom_in, Conn_j.bottom_ out} is a finite the set of nodes. Nodes represent the interface of component or connector, and component interface with a hollow circle, connector interface with a solid circle represents.
- $E \subseteq V \times V$ is a finite set of edges.
- $V_{start} \in$ {Comps_i.top_out | Comps_i.bottom_in = $\varnothing$ $\wedge$ Comps_i.bottom_out = $\varnothing$, $\text{Comps}_i \in$ Comps} is the initial node, this node transmit messages only.
- $V_{end} \in$ {Comps_i.bottom_in | Comps_i.top_out = $\varnothing$ $\wedge$ Comps_i.top_in = $\varnothing$, $\text{Comps}_i \in$ Comps} is the terminal node, this node receive messages only.

There are three types of edge in the CIG of a C2-style architecture specification, namely, edge from component to connector, edge from connector to component, and edge from connector to connector, which represents information flows between component and connector.

Definition 2.3 Let E is the edge set of CIG, element of E is divided into three edges.

- $e_{\text{Comps-Conn}}$ = {e | e $\in$ (Comps_i.top_out, Conn_j. bottom_in) $\vee$ (Comps_i.bottom_out, Conn_j.top_in)} represents edge from component Comps_i to connector Conn_j.
- $e_{\text{Conn-Comps}}$ = {e | e $\in$ (Conn_i.bottom_out, Comps_j. top_in) $\vee$ (Conn_i.top_out, Comps_j.bottom_in)} represents edge from connector Conn_i to component Comps_j.
- $e_{\text{Conn-Conn}}$ = {e | e $\in$ (Conn_i.top_out, Conn_j. bottom_in) $\vee$ (Conn_i.bottom_out, Conn_j.top_in)} represents edge from connector Conn_i to connector Conn_j.

CIG can be represented as an adjacency list. Adjacency node of each node in CIG can be represented as a linked list. Adjacency list consists of the order of the list storage nodes n and n a linked list. The order of the list for storage, each part contains two domains, one domain stores component or connector name, the other is pointer domain, point at adjacency list of CIG. The node of linked list consists of three domains, one represent as node V_i and serial number of V_j adjacent node, one represent as point at node V_i and next node V_k adjacent node, the other represent as point at edge information between node V_i and node V_k.

CIG construction algorithm is shown as follows.

```
Algorithm 1 BuiltCIG
Input: Adjacency list of CIG = (V, E, $V_{start}$, $V_{end}$)
Output: Component interaction graph
Begin
  $V = \varnothing$; $E = \varnothing$; $V_{start} = \varnothing$; $V_{end} = \varnothing$;
  appoints a starting node $V_1$;
  visits to the starting node $V_1$;
  $V_{start} = V_1$;
  while ($V_1 \neq \varnothing$) {
    $V = V \cup \{V_{start}\}$;
```

```
        obtains node $V_1$ and $V_2$ adjacent node from adjacency list;
        while ($V_2 \neq \varnothing$) {
          visits an edge e between node $V_1$ and $V_2$;
          do{
            switch (type of e) {
              case $e_{Comps\text{-}Conn}$: {
                $E = E \cup \{e_{Comps\text{-}Conn}\}$;break;}
              case $e_{Conn\text{-}Comps}$: {
                $E = E \cup \{e_{Conn\text{-}Comps}\}$;break;}
              case $e_{Conn\text{-}Conn}$: {
                $E = E \cup \{e_{Conn\text{-}Conn}\}$;break;}
            }
            visits next an edge e between node $V_1$ and $V_2$;
          }while (e = $\varnothing$);
          $V_{end} = V_{end} \cup \{V_2\}$;
          visits next node $V_1$ and $V_2$ adjacent node;
        }
        visits next node $V_1$;
        $V_{start} = V_1$;
      }
    End Algorithm BuiltCIG
```

III. Edge Coverage Criteria and Algorithms

We model component interactions using CIG which depicts interaction scenarios among components. Coverage criteria require that a set of entities of the CIG is covered when the test cases are executed.

Definition 3.1 A set of test cases represented by TS = $\{t_1, t_2, \ldots, t_n\}$, where a test case is triple t_i = (Pre, In, Out), it can cause C2-style architecture to input a set of execution, Pre represents pre-condition of input in C2-style architecture, In represents of input value in C2-style architecture, Out represents of expected outputs in C2-style architecture.

Testing coverage criteria can be used in one of two ways, as a mechanism to help testers mechanically or manually generate tests (test generation), or to measure the quality of pre-existing tests (coverage analysis). Edge coverage criteria of CIG are described below.

A. Component to Connector Edge Coverage Criteria (EComps-ConnCC)

A test case set TS satisfies the component to connector edge coverage criteria if and only if for each $e_{Comps\text{-}Conn}$ is executed by t in TS. In CIG, the result of EComps-ConnCC can be formalized as follows:

($Comps_i$.top_out , $Conn_j$.bottom_in) ## or

($Comps_i$.bottom_out , $Conn_j$.top_in)

Algorithm 2 is used to create component to connector edge coverage set of CIG. The functions are explained as follows: If find an edge from component $Comps_i$ to connector $Conn_j$, then this edge will be added to the edge coverage set of the component to connector. m is numbers of component, n is numbers of connector. The algorithm 2 is shown as follows.

```
Algorithm 2 Finde$_{Comps\text{-}Conn}$
Input: CIG
Output: Edge coverage set from component to connector
Begin
  EdgeComps_ConnSet = $\varnothing$; eComps_Conn = $\varnothing$;
  for (i = 1; i < = m; i++ )
    for (j = 1; j < = n; j++ ) {
      if ((comps[i].top_out, conn[j].bottom_in) $\in$ E){
        eComps_Conn = ## (comps[i].top_out, conn[j].bottom_in) ##;
        EdgeComps_ConnSet = EdgeComps_ConnSet $\cup$ eComps_Conn;}
      if ((comps[i].bottom_out, conn[j].top_in) $\in$ E){
        eComps_Conn = ## (comps[i].bottom_out, conn[j].top_in) ##;
        EdgeComps_ConnSet = EdgeComps_ConnSet $\cup$ eComps_Conn;}}
End Algorithm Finde$_{Comps\text{-}Conn}$
```

B. Connector to Component Edge Coverage Criteria (EConn-CompsCC)

A test case set TS satisfies the connector to component edge coverage criteria if and only if for each $e_{Conn\text{-}Comps}$ is executed by t in TS. In CIG, the result of EConn-CompsCC can be formalized as follows:

($Conn_i$.bottom_out , $Comps_j$.top_in) ## or

($Conn_i$.top_out , $Comps_j$.bottom_in)

Algorithm 3 is used to create connector to component edge coverage set of CIG. The functions are explained as follows: If find an edge from connector $Conn_i$ to component $Comps_j$, then this edge will be added to the edge coverage set of the connector to component. m is numbers of component, n is numbers of connector. The algorithm 3 is shown as follows.

```
Algorithm 3 Finde$_{Conn\text{-}Comps}$
Input: CIG
Output: Edge coverage set from connector to component
Begin
  EdgeConn_CompsSet = $\varnothing$; eConn_Comps = $\varnothing$;
  for (i = 1; i < = n; i++ )
    for (j = 1; j < = m; j++ ) {
      if ((conn[i].bottom_out, comps[j].top_in) $\in$ E){
        eConn_Comps = ## (conn[i].bottom_out, comps[j].top_in) ##;
        EdgeConn_CompsSet = EdgeConn_CompsSet $\cup$ eConn_Comps;}
      if ((conn[i].top_out, comps[j].bottom_in) $\in$ E){
        eConn_Comps = ## (conn[i].top_out, comps[j].bottom_in) ##;
        EdgeConn_CompsSet = EdgeConn_CompsSet $\cup$ eConn_Comps;}}
End Algorithm Finde$_{Conn\text{-}Comps}$
```

C. Connector to Connector Edge Coverage Criteria (EConn-ConnCC)

A test case set TS satisfies the connector to connector edge coverage criteria if and only if for each $e_{Conn\text{-}Conn}$ is executed by t in TS. In CIG, the result of EConn-ConnCC can be formalized as follows:

($Conn_i$.top_out , $Conn_j$.bottom_in) ## or

($Conn_i$.bottom_out , $Conn_j$.top_in)

Algorithm 4 is used to create connector to connector edge coverage set of CIG. The functions are explained as follows: If find an edge from connector Conn_i to connector Conn_j, then this edge will be added to the edge coverage set of the connector to connector. n is numbers of connector. The algorithm 4 is shown as follows.

```
Algorithm 4 Finde_{Conn-Conn}
Input: CIG
Output: Edge coverage set from connector to connector
Begin
  EdgeConn_ConnSet = ∅; eConn_Conn = ∅;
  for (i = 1; i < = n; i++ )
    for (j = 1; j < = n; j++ ) {
      if (conn[i].top_out, conn[j].bottom_in)∈E){
        eConn_Conn = ## (conn[i].top_out, conn[j].bottom_in) ##;
        EdgeConn_ConnSet = EdgeConn_ConnSet ∪ eConn_Conn;}
      if (conn[i].bottom_out, conn[j].top_in)∈E){
        eConn_Conn = ## (conn[i].bottom_out, conn[j].top_in) ##;
        EdgeConn_ConnSet = EdgeConn_ConnSet ∪ eConn_Conn;}}
End Algorithm Finde_{Conn-Conn}
```

Ⅳ. Edge Coverage Analysis

Edge coverage analysis is a structural testing technique, which helps to eliminate gaps in a test suite and determines when to stop testing. We use four metrics standard to evaluate the effectiveness of edge coverage criteria.

Let $|Comps|$ is number of component of C2-style architecture, $|Conn|$ is number of connector of C2-style architecture, $|e_{Comps-Conn}|$ is the number of edge from component to connector, $|e_{Conn-Comps}|$ is the number of edge from connector to component, $|e_{Conn-Conn}|$ is the number of edge from connector to connector.

Definition 4.1 The coverage of component to connector is the total of edge from component to connector divided by the number of component and connector in C2-style architecture. It is defined as follows:

$$EC_{Comps}^{Conn} = \frac{\sum_{i=1}^{|Comps|}\sum_{j=1}^{|Conn|} |e_{Comps_i - Conn_j}|}{|Comps| + 2|Conn|} \times 100\% \qquad (1)$$

Definition 4.2 The coverage of connector to component is the total of edge from connector to component divided by the number of component and connector in C2-style architecture. It is defined as follows:

$$EC_{Conn}^{Comps} = \frac{\sum_{i=1}^{|Conn|}\sum_{j=1}^{|Comps|} |e_{Conn_i - Comps_j}|}{|Comps| + 2|Conn|} \times 100\% \qquad (2)$$

Definition 4.3 The coverage of connector to connector is the total of edge from connector to connector divided by the number of component and connector in C2-style architecture. It is defined as follows:

$$EC_{Conn}^{Conn} = \frac{\sum_{i=1}^{|Conn|}\sum_{j=1}^{|Conn|} |e_{Conn_i - Conn_j}|}{|Comps| + 2|Conn|} \times 100\% \qquad (3)$$

Definition 4.4 The coverage of C2-style architecture is the average of the coverage of component to connector, the coverage of connector to component, and the coverage of connector to connector. It is defined as follows:

$$EC^{C2} = \frac{EC_{Comps}^{Conn} + EC_{Conn}^{Comps} + EC_{Conn}^{Conn}}{3} \qquad (4)$$

Ⅴ. Case Study

In order to better describe the above modeling process and explain the correctness of analysis process, we have implemented our proposed technique for the computation of edge coverage for simple JAVA programs. The experiment results, results analysis, and discussion are described in detail.

A. Case

KLAX system [3] is a video game of the C2-style architectural. The game includes three parts: KLAX Chute drops tiles of random colors at random times and locations; KLAX Palette catches tiles coming down the Chute and drops them into the Well, and Well removes horizontal, vertical, and diagonal sets of three or more consecutive tiles of the same color the collapse down the tiles above them to fill in the newly-created empty spaces.

The game calculates the scores accordingly. The C2-style architecture designed for the game of KLAX is shown in Fig. 3, where ADT components encapsulate the game's state, Logic components request changes of ADT state in accordance with game rules and interpret ADT state change notifications to determine the state of the game in progress, and artist components maintain the state of a set of abstract graphical objects.

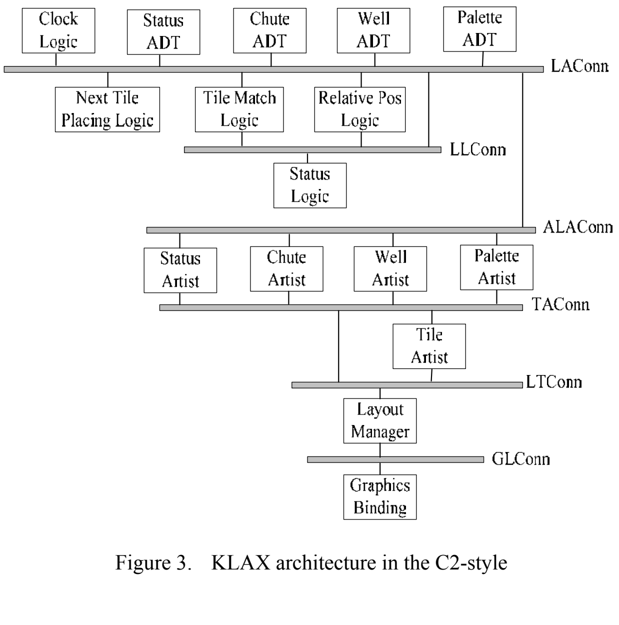


Figure 3. KLAX architecture in the C2-style

Fig. 4 shows the corresponding CIG for the example KLAX system of Fig. 3 according to C2-style architecture specification. In the figure, there are three types of edge, which are (LM.top_out, LTC.bottom_in) represents the edge from component LayoutManager to connector LTConn, (ALAC.bottom_out, PA.top_in) represents the edge from connector ALAConn to component PaletteADT, and (LTC.top_out, TAC. bottom_in) represents the edge from connector LTConn to connector TAConn.

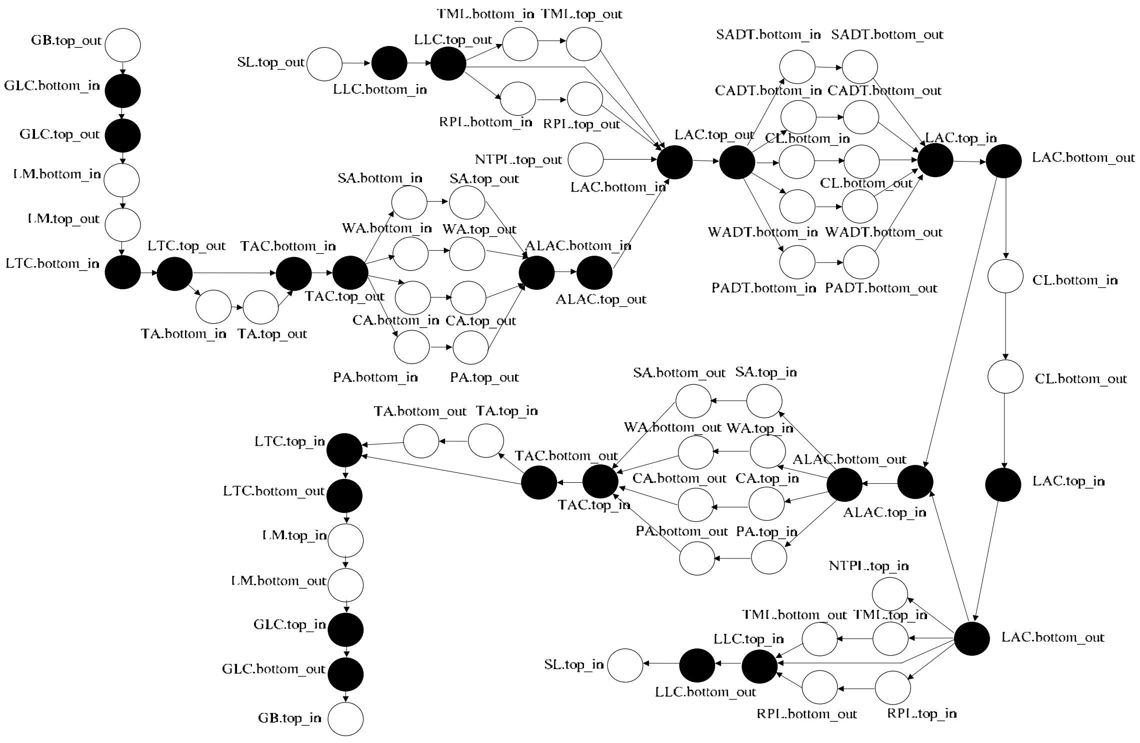


Figure 4. CIG of KLAX system

For example, the CIG depicted in Fig. 4 has:

V = {GraphicsBinding.top_out, GraphicsBinding.top_in, GLConn.bottom_out, GLConn.bottom_in, GLConn. top_out, GLConn.top_in, LayoutManager.bottom_out, LayoutManager.bottom_in, LayoutManager.top_out, LayoutManager.top_in, …}.

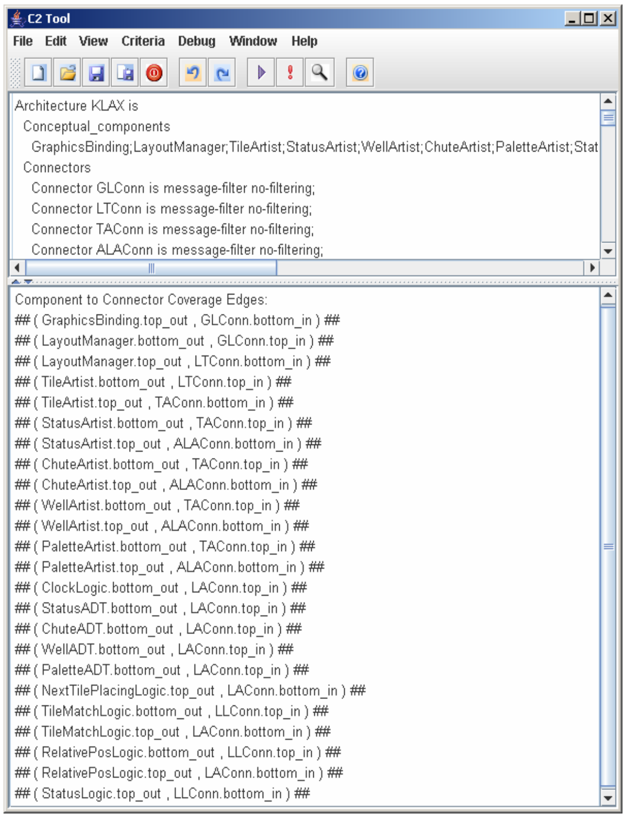
E = {(GraphicsBinding.top_out, GLConn.bottom_in), (GLConn.bottom_out, GraphicsBinding.top_in), (GLConn.top_out, LayoutManager.bottom_in), …}.

V_{start} = {GraphicsBinding.top_out, NextTilePlacing Logic.top_out, StatusLogic.top_out}.

V_{end} = {ClockLogic.bottom_in, StatusADT.bottom_in, ChuteADT.bottom_in, WellADT.bottom_in, PaletteADT. bottom_in}.

B. Experiment Result

We have implemented a prototype tool named C2 Tool that generates edge coverage automatically by our approach. We have implemented our tool using JAVA programs. The tool uses C2-ADL specification as input. Then it analyzes the names of all components, connectors and interfaces, interfaces types, the connection relationship between components and connectors and so on. These are stored in corresponding data structure respectively. Then according to edge coverage criteria, it can generate edge coverage set. In addition, the tool also provides the help documents about the details of the system functions, the operation and some open source code in an html format and so on.



```
Architecture KLAX is
  Conceptual_components
    GraphicsBinding;LayoutManager;TileArtist;StatusArtist;WellArtist;ChuteArtist;PaletteArtist;Stat
  Connectors
    Connector GLConn is message-filter no-filtering;
    Connector LTConn is message-filter no-filtering;
    Connector TAConn is message-filter no-filtering;
    Connector ALAConn is message-filter no-filtering;
```

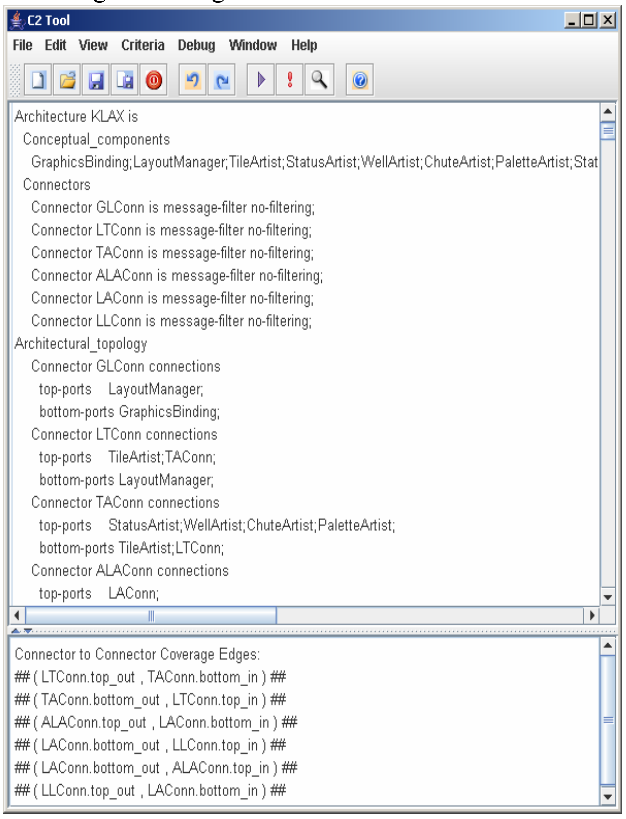
```
Component to Connector Coverage Edges:
## ( GraphicsBinding.top_out , GLConn.bottom_in ) ##
## ( LayoutManager.bottom_out , GLConn.top_in ) ##
## ( LayoutManager.top_out , LTConn.bottom_in ) ##
## ( TileArtist.bottom_out , LTConn.top_in ) ##
## ( TileArtist.top_out , TAConn.bottom_in ) ##
## ( StatusArtist.bottom_out , TAConn.top_in ) ##
## ( StatusArtist.top_out , ALAConn.bottom_in ) ##
## ( ChuteArtist.bottom_out , TAConn.top_in ) ##
## ( ChuteArtist.top_out , ALAConn.bottom_in ) ##
## ( WellArtist.bottom_out , TAConn.top_in ) ##
## ( WellArtist.top_out , ALAConn.bottom_in ) ##
## ( PaletteArtist.bottom_out , TAConn.top_in ) ##
## ( PaletteArtist.top_out , ALAConn.bottom_in ) ##
## ( ClockLogic.bottom_out , LAConn.top_in ) ##
## ( StatusADT.bottom_out , LAConn.top_in ) ##
## ( ChuteADT.bottom_out , LAConn.top_in ) ##
## ( WellADT.bottom_out , LAConn.top_in ) ##
## ( PaletteADT.bottom_out , LAConn.top_in ) ##
## ( NextTilePlacingLogic.top_out , LAConn.bottom_in ) ##
## ( TileMatchLogic.bottom_out , LLConn.top_in ) ##
## ( TileMatchLogic.top_out , LAConn.bottom_in ) ##
## ( RelativePosLogic.bottom_out , LLConn.top_in ) ##
## ( RelativePosLogic.top_out , LAConn.bottom_in ) ##
## ( StatusLogic.top_out , LLConn.bottom_in ) ##
```

Figure 5. C2 tool application interface and EComps-ConnCC set

Fig. 5 shows the application interface of the C2 tool and the edge coverage set from component to connector according to coverage criteria EComps-ConnCC, where the upper part represents C2-ADL specification of software architecture, the next part is edge coverage set according to coverage criteria correspondingly.

It can be discovered from Fig. 5 that coverage criteria EComps-ConnCC covers 24 edges from component to connector according to KLAX system specification. Similar, coverage criteria EConn-CompsCC covers 24 edges from connector to component.

Fig. 6 shows the application interface of the C2 tool and the edge coverage set from connector to connector according to coverage criteria EConn-ConnCC.



```
Architecture KLAX is
  Conceptual_components
    GraphicsBinding;LayoutManager;TileArtist;StatusArtist;WellArtist;ChuteArtist;PaletteArtist;Stat
  Connectors
    Connector GLConn is message-filter no-filtering;
    Connector LTConn is message-filter no-filtering;
    Connector TAConn is message-filter no-filtering;
    Connector ALAConn is message-filter no-filtering;
    Connector LAConn is message-filter no-filtering;
    Connector LLConn is message-filter no-filtering;
Architectural_topology
    Connector GLConn connections
      top-ports    LayoutManager;
      bottom-ports GraphicsBinding;
    Connector LTConn connections
      top-ports    TileArtist;TAConn;
      bottom-ports LayoutManager;
    Connector TAConn connections
      top-ports    StatusArtist;WellArtist;ChuteArtist;PaletteArtist;
      bottom-ports TileArtist;LTConn;
    Connector ALAConn connections
      top-ports    LAConn;
```

```
Connector to Connector Coverage Edges:
## ( LTConn.top_out , TAConn.bottom_in ) ##
## ( TAConn.bottom_out , LTConn.top_in ) ##
## ( ALAConn.top_out , LAConn.bottom_in ) ##
## ( LAConn.bottom_out , LLConn.top_in ) ##
## ( LAConn.bottom_out , ALAConn.top_in ) ##
## ( LLConn.top_out , LAConn.bottom_in ) ##
```

Figure 6. C2 tool application interface and EConn-ConnCC set

Tab. I illustrate the computation of three coverage using the Fig. 3 and experiment result of Fig. 5 and Fig. 6.

TABLE I.
COVERAGE COMPUTATIONS ON KLAX

Coverage	Computation	Value
EC_{Comps}^{Conn}	$=\dfrac{8\times 1+8\times 2}{16+2\times 6}$	$=85.7\ \%$
EC_{Conn}^{Comps}	$=\dfrac{2\times 2+5+2\times 4+3\times 1+4}{16+2\times 6}$	$=85.7\ \%$
EC_{Conn}^{Conn}	$=\dfrac{1+1+1+1+1+1}{16+2\times 6}$	$=21.4\ \%$

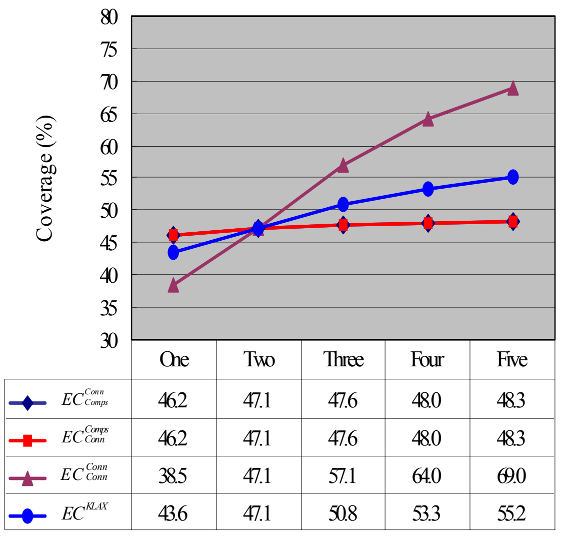
According to (4), the coverage result of KLAX system is:

$$EC^{KLAX}=\frac{1}{3}(85.7\%+85.7\%+21.4\%)=64.3\%$$

C. Result Analysis

Fig. 7 shows the Elevator system [4] experiment results [5]. In this figure, X-coordinate describes the Elevator car scales of the Elevator system: there is one, two, three, four, and five Elevator car respectively. Y-coordinate is the test coverage.

There are four curves in this figure: Three is the result according to methods proposed in this paper. The other is the coverage result of C2-style architecture. The values in this figure corresponding to the same X-value are obtained under the same Elevator car.



	One	Two	Three	Four	Five
◆ EC_{Comps}^{Conn}	46.2	47.1	47.6	48.0	48.3
■ EC_{Conn}^{Comps}	46.2	47.1	47.6	48.0	48.3
▲ EC_{Conn}^{Conn}	38.5	47.1	57.1	64.0	69.0
● EC^{KLAX}	43.6	47.1	50.8	53.3	55.2

Figure 7. Experiment result

From Fig. 7, we have:

- The coverage of component to connector is same as the coverage of connector to component, and the value increased continuously with the Elevator car growing.
- The coverage of connector to connector increased continuously with the Elevator car growing.
- The coverage of component to connector and the coverage of connector to connector have a result of the same with the Elevator car growing.
- The coverage of C2-style architecture increased continuously with the Elevator car growing.

D. Discussion

Zhenyi and Offutt defined six architecture relations [6] among architecture units: Component(Connector)_Internal_Transfer_Relation(N.interf$_1$, N.interf$_2$), Component(Connector)_Internal_Sequencing_Relation(N.interf$_1$, N.interf$_2$), Component(Connector)_Internal_Relation(N$_1$.interf$_1$, N$_1$.interf$_2$), N_C_Relation(N.interf$_1$, C.interf$_1$) or C_N_Relation(C.interf$_1$, N.interf$_1$), Direct_Component_Relation(N$_1$.interf$_1$, C$_1$.interf$_1$, C$_1$.interf$_2$, N$_2$.interf$_2$), and Indirect_Component_Relation(N$_1$.interf$_1$, C$_1$.interf$_1$, C$_1$.interf$_2$, N$_2$.interf$_2$, C$_2$.interf$_1$, C$_2$.interf$_2$, N$_3$.interf$_1$). The relations are used to define architecture testing paths, which are then used to define architecture level testing criteria. Let N$_i$ are components, C$_j$ are connectors, and Interf$_k$ are interfaces. Where:

- N_C_Relation(N.interf$_i$, C.interf$_j$) is equivalent to $e_{Comps\text{-}Conn}$(N.top_out, C.bottom_in) or $e_{Comps\text{-}Conn}$ (N.bottom_out, C.top_in), C_N_Relation (C.interf$_i$, N.interf$_j$) is equivalent to $e_{Conn\text{-}Comps}$(C. top_out, N.bottom_in) or $e_{Conn\text{-}Comps}$(C.bottom_ out, N.top_in).
- Direct_Component_Relation(N_1.interf$_i$, C_2.interf$_j$) and a C_N_Relation(C_2.interf$_k$, N_3.Interf$_l$) is equivalent to the combination of $e_{Comps\text{-}Conn}$(N_1. top_out, C_2.bottom_in) and $e_{Conn\text{-}Comps}$(C_2.top_out, N_3.bottom_in) or $e_{Comps\text{-}Conn}$(N_1.bottom_out, C_2. top_in) and $e_{Conn\text{-}Comps}$(C_2.bottom_out, N_3.top_in).
- If there are relations that connect N_1, N_2, N_3, C_1, and C_2 together, then the resulting path N_1–C_1–N_2–C_2–N_3 is equivalent to a number of combinations of $e_{Comps\text{-}Conn}$(N_1.top_out, C_1. bottom_in), $e_{Conn\text{-}Comps}$(C_1.top_out, N_2.bottom_in), $e_{Comps\text{-}Conn}$(N_2.top_out, C_2.bottom_in), and $e_{Conn\text{-}Comps}$(C_2.top_out, N_3.bottom_in) or $e_{Comps\text{-}Conn}$(N_1. bottom_out, C_1.top_in), $e_{Conn\text{-}Comps}$(C_1.bottom_ out, N_2.top_in), $e_{Comps\text{-}Conn}$(N_2.bottom_out, C_2. top_in), and $e_{Conn\text{-}Comps}$(C_2.bottom_out, N_3.top_in).

Ⅵ. Related Work

The technologies of software architecture testing mainly concentrate on the establishment of abstract testing model and the extraction of dynamic architecture features. Bertolino et al [1,7] proposed the thought that testing cases were derived from the software architecture description, founding the beginning of a matter for software architecture analyzing and testing.

Muccini et al [8] used software architecture as the reference model for testing with its corresponding architecture specification. Zhenyi and Offutt presented in [6] an architecture-based testing technique to test software. Software architectures abstract away details from applications so the applications can be viewed as sets of components with connectors that describe the interactions among components. Architecture description languages are used to model software architecture for analysis and development. Muccini et al [9] proposed an approach to handle the retesting of a software system during evolution of both its architecture and implementation, while reducing the testing effort.

Lun and Xu introduced πBG to describe software architecture. They presented seven testing coverage criteria and discussed the subsume relation between testing criteria [10,11]. Lun and Ding [12] presented a formal approach to analyze architecture-based test criteria by RDG and automata based on the formal description of the software architecture, and generating test sequences according to two testing criteria. Lun and Chi [13] presented a software architecture testing technology base on C2-style. First, it describes software architecture through C2-style, then represents software architecture through CIG, and abstracted the behavior of interactive between components and connectors. Formalized architecture edge coverage, generated the testing edge sets that covered the architecture according to the edge coverage criterion and algorithms, and analyze the relation of test path numbers between software architecture testing criteria. Muccini et al [14] proposed a specialization and refinement of our general approach for SA-based conformance testing, he deal with the SA to code mapping rules imposed by the C2 framework helps to limit the mapping problem, and allows a systematic testing approach.

Ⅶ. Conclusions and Future Work

This paper presents a software architecture testing technology based on C2-style. First, it describes software architecture through C2-style, then represents software architecture through CIG, and abstracted the behavior of interactive between components and connectors. Formalized architecture edge coverage, generated the testing edge sets that covered the architecture according to the edge coverage criterion and algorithms. This technology could establish an abstract model to describe the characteristics of dynamic architecture, it covered all the testing component nodes and reduced scale of testing coverage set, so that test the architecture effectively.

As for the future work, the application of the approach needs to study at the implementation level. It is also planned to investigate other testing criteria and testing criteria adequacy and the approach to generate test cases which satisfy the testing criteria without necessarily simulating the execution process of all possible test paths.

Acknowledgment

Part of this work is supported by the Natural Science Foundation of Heilongjiang Province of China under Grant No. F201036, the Scientific Research Foundation of Heilongjiang Provincial Education Department of China under Grant No. 11551127.

References

[1] A. Bertolino, P. Inverardi, H. Muccini, “An Explorative Journey from Architectural Tests Definition downto Code Test Execution”, *in: Proc. of 23rd ACM/IEEE International Conference on Software Engineering (ICSE2001)*, Washington, USA, pp. 211-220, May 2001.

[2] N. T. Richard, N. Medvidovic, K. M. Anderson, E. J. Whitehead, J. E. Robbins, “A Component- and Message-Based Architecture Style for GUI Software”, *IEEE Trans. on Software Engi.*, vol. 22, pp. 390-406, June 1996.

[3] M. M. Gorlick, R. R. Razouk, “Using Weaves for Software Construction and Analysis”, *in: Proc. of 13th Int’1. Conf. Software Engineering*, Los Alamitos, CA, USA, pp. 23-34, May 1991.

[4] H. Muccini, M. Dias, D. J. Richardson, “Software Architecture based Regression Testing”, *Journal of Systems and Software*, vol. 79, pp. 1379-1396, October 2006.

[5] L. J. Lun, X. Chi, “On the Relation of Software Architecture Testing Criteria in the C2 Style”, *in: Proc. of 2nd of the International Conference on Computational Intelligence and Software Engineering (CiSE2010)*, Wuhan, China, December 2010.

[6] J. Zhenyi, J. Offutt, “Deriving Tests from Software Architectures”, *in: Proc. of 12th IEEE International*

Symposium on Software Reliability Engineering, Washington, DC, USA, pp. 308-313, November 2001.

[7] A. Bertolino, F. Corradini, P. Inverardi, H. Muccini, "Deriving Test Plans from Architectural Descriptions", *in: Proc. of ACM Proceedings International Conference on Software Engineering (ICSE2000)*, New York, USA, pp. 220-229, June 2000.

[8] H. Muccini, A. Bertolino, P. Inverardi, "Using Software Architecture for Code Testing", *IEEE Trans. on Software Engi.*, vol. 30, pp. 160-171, March 2004.

[9] H. Muccini, M. Dias, D. J. Richardson, "Towards Software Architecture-based Regression Testing", *ACM SIGSOFT Software Engineering Notes*, vol. 30, pp. 1-7, April 2005.

[10] L. J. Lun, H. Xu, "An Approach to Software Architecture Testing", *in: Proc. of 9th International Conference for Young Computer Scientists (ICYCS2008)*, Zhangjiajie, China, pp. 1070-1075, November 2008.

[11] L. J. Lun, H. Xu, "Analysis of the Subsume Relation between Software Architecture Testing Criteria", *in: Proc. of 2008 International Conference on Computer Science and Software Engineering (CSSE2008)*, Wuhan, China, pp. 698-701, December 2008.

[12] L. J. Lun, X. M. Ding, "Analyzing Relation between Software Architecture Testing Criteria on Test Sequences", *in: Proc. of 2009 IEEE Secure Software Integration and Reliability Improvement (SSIRI2009)*, Shanghai, China, pp. 181-186, July 2009.

[13] L. J. Lun, X. Chi, "Software Architecture Testing in the C2 Style", *in: Proc. of 2010 3rd International Conference on Advanced Computer Theory and Engineering (ICACTE2010)*, Chengdu, China, vol. 1, pp. 123-127, August 2010.

[14] H. Muccini, M. Dias, D. J. Richardson, "Systematic Testing of Software Architectures in the C2 style", *in: Conf. Fundamental Approaches to Software Engineering (FASE2004)*, Barcelona, Spain, LNCS 2984, pp. 295-309, March 2004.

Lijun Lun was born in Harbin, Heilongjiang Province, China, in 1963. He received his B.S. degree and Master degree in Computer Science and Technology from Harbin Institute Technology of Computer Science and Technology, China, in 1986 and 2000 respectively.

Currently, he is a professor, and teaches and conducts research in the areas of software architecture, software testing, and software metrics, etc.

Xin Chi was born in Harbin, Heilongjiang Province, China, in 1990. She is a three year's college student at Harbin Normal University, China, since 2009. She has been engaged in software architecture testing and software metrics research for approximately three years.

Xuemei Ding was born in Harbin, Heilongjiang Province, China, in 1972. She received her B.S. degree and Master degree in Computer Science and Technology from Heilongjiang University and Harbin Institute Technology of Computer Science and Technology, China, in 1996 and 2000 respectively. Currently, she is an associate professor, and teaches and conducts research in the areas of software engineering, neural network, and one-class classification, etc.