Computing and Informatics, Vol. 36, 2017, 386–404, doi: 10.4149/cai_2017_2_386

AUTOMATIC GENERATION OF BASIS COMPONENT PATH COVERAGE FOR SOFTWARE ARCHITECTURE TESTING

Lijun Lun, Shaoting Wang, Xin Chi

College of Computer Science and Information Engineering
Harbin Normal University
150080 Harbin, China
e-mail: lunlijun@yeah.net, 736620872@qq.com, xinc1990@163.com

Hui Xu

Library
Heilongjiang University of Chinese Medicine
150040 Harbin, China
e-mail: xuhui8413@163.com

Abstract. Architecture-centric development is one of the most promising methods for improving software quality, reducing software cost and raising software productivity. Software architecture research not only focuses on the design phase, but also covers every phase of software life cycle. Software architecture has characteristics different from traditional software, conventional testing methods do not apply directly to software architecture. Basis path testing is a very simple and efficient white-box testing method. Traditional methods generate basis path according to the control flow graph, they are not suitable for generating component path when we detect more software architecture errors. This paper presents a new concept – Basis Component Path (BCP) for C2-style architecture, and proposes a method to generate the BCPs. C2-style architecture is represented by components, connectors, and interfaces, and uses an architecture component interaction graph (CIG) to describe interface connection relationship. We also provide an algorithm to generate BCP set. Experiments apply the proposed method in a typical C2-style architecture and the result shows that the proposed method can generate BCP set which contains as many BCPs as possible efficiently, and it meets the requirements of the basis component path testing.

Keywords: Software architecture testing, C2-style architecture, component interaction graph, basis component path generation

Mathematics Subject Classification 2010: 68N30

1 INTRODUCTION

Software architecture [1] 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 software development life cycle. Software architectures are nowadays used for different purposes, including driving analysis techniques [2]. Software architecture testing is an important technique for validating and checking the correctness of software architecture. Formalization testing [3, 4] based on software architecture has improved the quality of the software products. Automatic test coverage generation is a hotspot and a difficulty in the field of software architecture testing [5]. Current research is divided into two categories [6]. 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 testing techniques and methods, so that it can better solve problems of software architecture testing.

The software architecture is the foundation for any software system. It represents the earliest design decisions that are both the most difficult to get right and the hardest to change downstream.

Path coverage is one of the most important criteria that investigate the sufficiency of software testing. It requires that every path in a program should be executed at least once. Basis path testing is a structural testing technique. It is a technology of reducing the scale of path testing, which reduces not only test operation amount but also the repeated generation of test suite.

This paper tries to focus on simplification of the CIG in order to generate the BCP, how many BCPs are required? The reason why we consider such a question is as follows.

First, a component has to take help of other component(s) to perform its functionality. In this way, testing activity between components is a time- and labor-consuming process. Test requirements can be used as a well accepted measure for selecting test suites, reducing test suites and deciding when to stop testing. In this paper, BCP can be regarded as test requirements. When we obtain the BCPs required, the costs for testing the C2-style architecture application will be reduced.

Secondly, the selection of test suites should guarantee that each test requirement is satisfied by at least one test suite. Thus, test requirement reduction can help to reduce the number of test suites and avoid redundant test suites.

So, we propose the BCP to describe the interaction relationships between components, and propose algorithm to generate BCP. We apply our method to a typical C2-style architecture, and experiments show that our method can generate BCP.

2 BACKGROUND

We will give some backgrounds in this section, mainly about some notations and the basis component path testing.

2.1 C2-Style Architecture

The C2-style architecture is primarily concerned with high-level system composition issues [7]. The C2-style architecture 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, 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 architecture, 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.

We represent a component as $Comp_i$, where $Comp_i \in Comp$ indicates i^{th} component of C2-style architecture and $Comp$ represents a set of components. I_p represents the set of interfaces of component, it consists of the set of top interfaces I_{pt} and the set of bottom interfaces I_{pb}. I_{pt} consists of the top output interface I_{pt_o} and top input interface I_{pt_i}, I_{pb} consists of the bottom output interface I_{pb_o} and bottom input interface I_{pb_i}. $Comp_i$ through interface $Comp_i.I_{pt\_o}$ or $Comp_i.I_{pb\_o}$ to send an event request, and through interface $Comp_i.I_{pt\_i}$ or $Comp_i.I_{pb\_i}$ to receive other component or connector to send messages. If $Comp_i$ does not have the top or bottom output interface, then $Comp_i$ cannot send messages from its top or bottom output interface. Similarly, if $Comp_i$ does not have the top or bottom input interface, then $Comp_i$ cannot receive messages from its top or bottom input interface.

We represent a connector as $Conn_j$, where $Conn_j \in Conn$ indicates j^{th} connector of C2-style architecture and $Conn$ represents a set of connectors. I_n represents the set of interfaces of connector, it consists of the set of top interfaces I_{nt} and the set of bottom interfaces I_{nb}. I_{nt} consists of the top output interface I_{nt_o} and top input interface I_{nt_i}, I_{nb} consists of the bottom output interface I_{nb_o} and bottom input interface I_{nb_i}. $Conn_j$ through interface $Conn_j.I_{nt\_o}$ or $Conn_j.I_{nb\_o}$ to send an event request, and through interface $Conn_j.I_{nt\_i}$ or $Conn_j.I_{nb\_i}$ to receive other component or connector to send messages.

We represent constraint as $Comp_i.I_{pt\_o} \rightarrow Conn_j.I_{nb\_i}$ means that there exists path from the top output interface of component $Comp_i$ to the bottom input interface of connector $Conn_j$. On the other hand, $Comp_i.I_{pb\_o} \rightarrow Conn_j.I_{nt\_i}$ means that there exists path from the bottom output interface of component $Comp_i$ to

the top input interface of connector $Conn_j$. $Conn_i.I_{nt}\_o \rightarrow Comp_j.I_{pb}\_i$ means that there exists path from the top output interface of connector $Conn_i$ to the bottom input interface of component $Comp_j$. $Conn_i.I_{nb}\_o \rightarrow Comp_j.I_{pt}\_i$ means that there exists path from the bottom output interface of connector $Conn_i$ to the top input interface of component $Comp_j$. $Conn_i.I_{nt}\_o \rightarrow Conn_j.I_{nb}\_i$ means that there exists path from the top output interface of connector $Conn_i$ to the bottom input interface of connector $Conn_j$. $Conn_i.I_{nb}\_o \rightarrow Conn_j.I_{nt}\_i$ means that there exists path from the bottom output interface of connector $Conn_i$ to the top input interface of connector $Conn_j$.

2.2 Component Interaction Graph

The C2-style architecture control flow is usually represented by a digraph. We use the CIG model to represent interaction relationships between interface of components and interface of connectors, and between interface of connectors.

Definition 1 (CIG)**.** CIG is a digraph $CIG = \langle V, E, V_{start}, V_{end}\rangle$, where V is the set of nodes, which represent the interface of the components and the interface of connectors, that is $\forall Comp_i \in Comp \wedge Conn_j \in Conn$, $V = \{Comp_i.I_{pt}\_i, Comp_i.I_{pt}\_o, Comp_i.I_{pb}\_i, Comp_i.I_{pb}\_o, Conn_j.I_{nt}\_i, Conn_j.I_{nt}\_o, Conn_j.I_{nb}\_i, Conn_j.I_{nb}\_o\}$. The interface of component nodes are expressed with hollow circular and the interface of connector nodes are expressed with solid circular. $E \subseteq V \times V$ is the set of edges, which represent the message interaction relationships between interface of components and interface of connectors, and between interface of connectors. $V_{start} = \{Comp_i.I_{pt}\_o \mid Comp_i.I_{pb}\_i = \emptyset,\ Comp_i \in Comp\}$ is the initial node, this node transmit messages only. $V_{end} = \{Comp_i.I_{pb}\_i \mid Comp_i.I_{pt}\_o = \emptyset, Comp_i \in Comp\}$ is the terminal node, this node receive messages only.

There are three types of edges 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 (Edge)**.** Given a component interaction graph $CIG = \langle V, E, V_{start}, V_{end}\rangle$, where $V = \{Comp_i.I_{pt}\_i, Comp_i.I_{pt}\_o, Comp_i.I_{pb}\_i, Comp_i.I_{pb}\_o, Conn_j.I_{nt}\_i, Conn_j.I_{nt}\_o, Conn_j.I_{nb}\_i, Conn_j.I_{nb}\_o\}$ and $E \subseteq V \times V$. If there exists $(Comp_i.I_{pt}\_o, Conn_j.I_{nb}\_i) \vee (Comp_i.I_{pb}\_o, Conn_j.I_{nt}\_i) \in E$, then $(Comp_i.I_{pt}\_o, Conn_j.I_{nb}\_i)$ and $(Comp_i.I_{pb}\_o, Conn_j.I_{nt}\_i)$ are edge from component to connector, called $e_{Comp-Conn}$ for short. If there exists $(Conn_j.I_{nt}\_o, Comp_i.I_{pb}\_i) \vee (Conn_j.I_{nb}\_o, Comp_i.I_{pt}\_i) \in E$, then $(Conn_j.I_{nt}\_o, Comp_i.I_{pb}\_i)$ and $(Conn_j.I_{nb}\_o, Comp_i.I_{pt}\_i)$ are edge from connector to component, called $e_{Conn-Comp}$ for short. If there exists $(Conn_i.I_{nt}\_o, Conn_j.I_{nb}\_i) \vee (Conn_i.I_{nb}\_o, Conn_j.I_{nt}\_i) \in E$, then $(Conn_i.I_{nt}\_o, Conn_j.I_{nb}\_i)$ and $(Conn_i.I_{nb}\_o, Conn_j.I_{nt}\_i)$ are edge from connector to connector, called $e_{Conn-Conn}$ for short.

In order to illustrate the C2-style architecture, we consider a simple C2-style architecture in Figure 1. There are seven components C_i $(i = 1, \ldots, 7)$ and four connectors B_j $(j = 1, \ldots, 4)$.

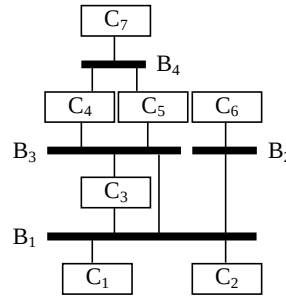

Figure 1. A simple C2-style architecture

The CIG of Figure 1 can be seen in Figure 2, there are 36 interfaces, for example, $C_2.I_{pt}\_o$ is an interface that represents the top output of the component C_2, $B_3.I_{nb}\_i$ is an interface that represents the bottom input of the connector B_3. The top interface $C_1.I_{pt}\_o$ of component C_1 transmits messages to the bottom interface $B_1.I_{nb}\_i$ of connector B_1, the top interface $B_2.I_{nt}\_o$ of connector B_2 transmits messages to the bottom interface $C_6.I_{pb}\_i$ of component C_6, and the bottom interface $C_7.I_{pb}\_o$ of component C_7 transmits messages to the top interface $B_4.I_{nt}\_i$ of connector B_4.

Path coverage is a kind of important standard that investigates the sufficiency of software testing. Component path coverage technology in software architecture is a structural testing method that involves using the architecture elements to attempt to find every possible executable path.

Definition 3 (Component path)**.** Given a component interaction graph $CIG = \langle V, E, V_{start}, V_{end} \rangle$, where $V_i.I_{pt}\_o, V_i.I_{pt}\_i, V_i.I_{pb}\_o, V_i.I_{pb}\_i, V_i.I_{nt}\_o, V_i.I_{nt}\_i, V_i.I_{nb}\_o, V_i.I_{nb}\_i \in V$ $(i = 1, 2, \ldots, k)$, $V_i \in Comp \cup Conn$ $(i = 1, 2, \ldots, k)$. A path from node V_1 to V_k is a sequence of nodes $V_1 \to V_2 \to \ldots \to V_k$, such that for $i = 1, 2, \ldots, k-1$:

- $(V_i.I_{pt}\_o, V_{i+1}.I_{nb}\_i) \vee (V_i.I_{nt}\_o, V_{i+1}.I_{pb}\_i) \vee (V_i.I_{nt}\_o, V_{i+1}.I_{nb}\_i) \in E$, or
- $(V_i.I_{pb}\_o, V_{i+1}.I_{nt}\_i) \vee (V_i.I_{nb}\_o, V_{i+1}.I_{pt}\_i) \vee (V_i.I_{nb}\_o, V_{i+1}.I_{nt}\_i) \in E$

if $V_1 \in Comp \wedge V_k \in Comp$, then the path is a component path for the CIG, called CP for short.

From the definition 3, we can see that the CP has two forms according to the type of edges, one is all edges from the beginning of top interface of component and connector to the end of bottom interface of component and connector, other is all edges from the beginning of bottom interface of component and connector to the end of top interface of component and connector.

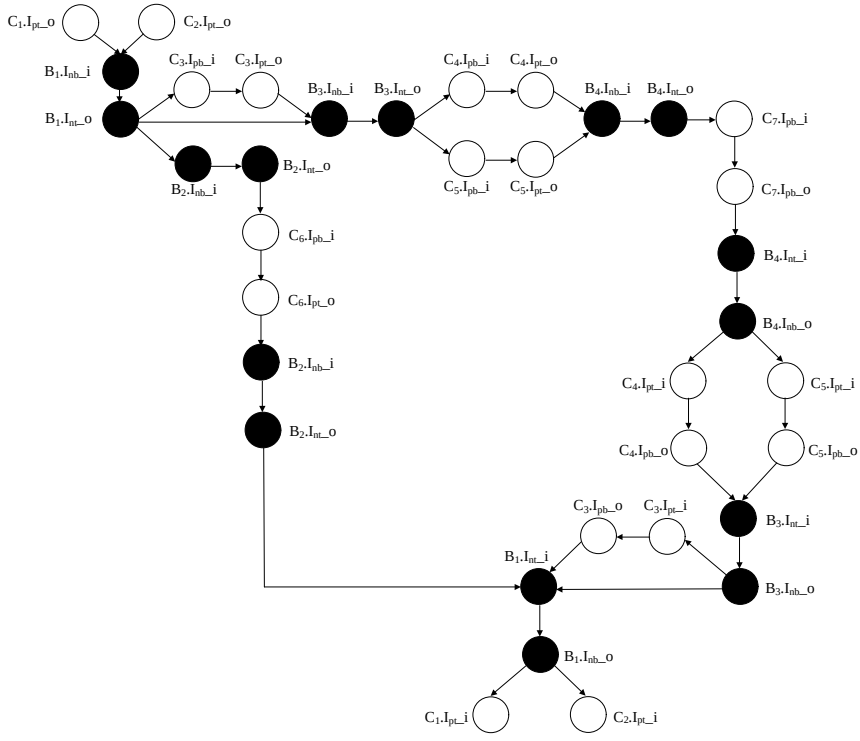

Figure 2. CIG of Figure 1

To understand concept of CP, let us consider again an example shown in Figure 2. We can see that $(C_1.I_{pt}\_o, B_1.I_{nb}\_i)$ is a $e_{Comp-Conn}$, $(B_1.I_{nt}\_o, C_3.I_{pb}\_i)$ is a $e_{Conn-Comp}$, $(C_3.I_{pt}\_o, B_3.I_{nb}\_i)$ is a $e_{Comp-Conn}$, $(B_3.I_{nt}\_o, C_4.I_{pb}\_i)$ is a $e_{Conn-Comp}$, $(C_4.I_{pt}\_o, B_4.I_{nb}\_i)$ is a $e_{Comp-Conn}$, $(B_4.I_{nt}\_o, C_7.I_{pb}\_i)$ is a $e_{Conn-Comp}$, $(B_1.I_{nt}\_o, B_3.I_{nb}\_i)$ is a $e_{Conn-Conn}$, $(B_3.I_{nt}\_o, C_5.I_{pb}\_i)$ is a $e_{Conn-Comp}$, and $(C_5.I_{pt}\_o, B_4.I_{nb}\_i)$ is a $e_{Comp-Conn}$. Thus there are four CPs from component C_1 to component C_7 are shown as follows.

- $C_1 \rightarrow B_1 \rightarrow C_3 \rightarrow B_3 \rightarrow C_4 \rightarrow B_4 \rightarrow C_7$.
- $C_1 \rightarrow B_1 \rightarrow C_3 \rightarrow B_3 \rightarrow C_5 \rightarrow B_4 \rightarrow C_7$.
- $C_1 \rightarrow B_1 \rightarrow B_3 \rightarrow C_4 \rightarrow B_4 \rightarrow C_7$.
- $C_1 \rightarrow B_1 \rightarrow B_3 \rightarrow C_5 \rightarrow B_4 \rightarrow C_7$.

Similarly, these two CPs from component C_4 to component C_2 are shown as follows.

- $C_4 \rightarrow B_3 \rightarrow C_3 \rightarrow B_1 \rightarrow C_2$.
- $C_4 \rightarrow B_3 \rightarrow B_1 \rightarrow C_2$.

CP means computing every single component path through the CIG. But, as the size and complexity of software system increases, the number of CPs gives rise to an infinite numbers of CPs, we cannot test all the CPs through a nontrivial software architecture. To reduce the number of CPs, we can test many partial component paths.

2.3 Basis Component Path

Basis path is a testing technique which is first introduced by McCabe [8]. Basis path testing enables the designer to derive a logical complexity measure of procedural design and then uses it as a guide for defining a basic set of execution paths. Test suites that exercise the basis set are guaranteed to execute every statement in the program at least once during testing. We define BCP according to the component path.

Definition 4 (Basis component path)**.** Given a component interaction graph $CIG = \langle V, E, V_{start}, V_{end} \rangle$, where $V_i.I_{pt\_}o, V_i.I_{pt\_}i, V_i.I_{pb\_}o, V_i.I_{pb\_}i, V_i.I_{nt\_}o, V_i.I_{nt\_}i, V_i.I_{nb\_}o, V_i.I_{nb\_}i \in V$ $(i = 1, 2, \ldots, k)$, $V_i \in Comp \cup Conn$ $(i = 1, 2, \ldots, k)$. If $V_1 \to V_2 \to \ldots \to V_k$ is the CP which covers all nodes from V_1 to V_k and at least one edge of this CP never appears in any other CP from V_1 to V_k, then the CP is a basis component path, called BCP for short. All of BCPs make up the basis component path set.

From the definition of BCP, a BCP can be differentiated from all other BCPs by at least one edge. Let us consider again an example shown in Figure 2. We can see that there are three BCPs from component C_1 to component C_7 are shown as follows.

- $C_1 \to B_1 \to C_3 \to B_3 \to C_4 \to B_4 \to C_7$.
- $C_1 \to B_1 \to C_3 \to B_3 \to C_5 \to B_4 \to C_7$.
- $C_1 \to B_1 \to B_3 \to C_4 \to B_4 \to C_7$.

McCabe [8] proved that the size of basis path set is unique for any given control flow graph and is called the cyclomatic complexity V(G) of the program. The cyclomatic complexity can be easily computed by the following formula:

$$V(G) = e - n + 2$$

where e is the number of edges in the control flow graph, n is the number of nodes in the control flow graph. The formula indicates that the cyclomatic complexity depends only on the structure of program.

Because there may be a BCP between any two component nodes, in order to make the cyclomatic complexity suitable for calculating the number of BCPs, we modify the cyclomatic complexity formula as the number of BCPs through its CIG and calculate it as follows:

$$|BCP_{V_i}^{V_j}| = \left(\sum_{V_i}^{V_j} |e_{Comp-Conn}| + \sum_{V_i}^{V_j} |e_{Conn-Comp}| + \sum_{V_i}^{V_j} |e_{Conn-Conn}| \right) - \left(\sum_{V_i}^{V_j} |n(Comp)| + \sum_{V_i}^{V_j} |n(Conn)| \right) + 2 \quad (1)$$

where $|BCP_{V_i}^{V_j}|$ represents the number of BCPs from component V_i to component V_j, $\sum_{V_i}^{V_j} |e_{Comp-Conn}|$ represents the number of edges from component to connector from V_i to V_j, $\sum_{V_i}^{V_j} |e_{Conn-Comp}|$ represents the number of edges from connector to component from V_i to V_j, $\sum_{V_i}^{V_j} |e_{Conn-Conn}|$ represents the number of edges from connector to connector from V_i to V_j, $\sum_{V_i}^{V_j} |n(Comp)|$ represents the number of components from V_i to V_j, and $\sum_{V_i}^{V_j} |n(Conn)|$ represents the number of connectors from V_i to V_j.

Consider the BCP from component C_1 to component C_4 of Figure 2. There are two edges from component to connector (C_1, B_1) and (C_3, B_3), two edges from connector to component (B_1, C_3) and (B_3, C_4), one edge connector to connector (B_1, B_3), three component nodes C_1, C_3, and C_4, and two connector nodes B_1 and B_3. Thus, the number of BCPs from C_1 to C_4 is: $|BCP_{C_1}^{C_4}| = (2+2+1)-(3+2)+2 = 2$, such that:

- BCP 1: $C_1 \rightarrow B_1 \rightarrow C_3 \rightarrow B_3 \rightarrow C_4$.
- BCP 2: $C_1 \rightarrow B_1 \rightarrow B_3 \rightarrow C_4$.

Thus, we can see that the number of BCPs is less than or equal to the number of component paths. We need a method to find the basis component path set.

3 BASIS COMPONENT PATH GENERATION METHOD

This section describes our proposed technique to generate BCP for C2-style architecture. The technique performs two tasks in two phases as follows:

- Generating the CIG according to the C2-style architecture specification.
- Generating the basis component path set based on CIG.

We give a detailed description of these two phases of the technique in the following subsections.

3.1 CIG Generation Phase

In order to generate CIG, we propose an algorithm GenCIG to generate CIG. The operation of our GenCIG algorithm outlines the different steps in constructing the CIG for each component and connector definition. First, the component and connector are created. Subsequently, edges are analyzed between component and connector.

Then, after performing components, connectors, and edges analysis, components, connectors, edges between component and connector are added.

We present the algorithm GenCIG of our proposed technique to generate the CIG is illustrated in Algorithm 1.

Algorithm 1 GenCIG

Input: C2-style architecture specification
Output: component interaction graph
begin
1: the architecture of each component and connector, an increase in the corresponding node;
2: **if** exists the message interaction from the interface I_p of component $Comp_1$ to the interface I_n of connector $Conn_2$ **then**
3: add an edge from I_p to I_n;
4: **end if**
5: **if** exists the message interaction from the interface I_n of connector $Conn_2$ to the interface I_p of component $Comp_1$ **then**
6: add an edge from I_n to I_p;
7: **end if**
8: **if** exists the message interaction from the interface I_{n_1} of connector $Conn_1$ to the interface I_{n_2} of connector $Conn_2$ **then**
9: add an edge from I_{n_1} to I_{n_2};
10: **end if**
11: **if** exists the message interaction from the interface I_{p_1} to I_{p_2} of component $Comp_1$ **then**
12: add a additional edge from I_{p_1} to I_{p_2} in $Comp_1$;
13: **end if**
14: **if** exists the message interaction from the interface I_{n_1} to I_{n_2} of connector $Conn_2$ **then**
15: add a additional edge from I_{n_2} to I_{n_1} in $Conn_2$;
16: **end if**
17: **return** CIG;
end

We now illustrate the construction of the CIG with the help of an example shown in Figure 2. The component C_1 and connector B_1 are created in line 1. The edges (C_1, B_1) and (C_4, B_4) are created in lines 2–4. The edges (B_1, C_3) and (B_2, C_6) are created in lines 5–7. The edges (B_1, B_2) and (B_3, B_1) are created in lines 8–10. The interfaces of C_5 and B_4 are created in lines 11–16. Finally, generate CIG (line 17).

3.2 BCP Set Generation Phase

In order to generate BCP set, we propose an algorithm BCPA to generate BCP set. The operation of our BCPA algorithm outlines the different steps in construct-

ing the basis component path set. The basis component path set is associated with an empty set before applying the algorithm (line 1). Then generate basis component path set (lines 2–34), where edge graph (EG) is constructed by the CIG to determine whether the generating component paths in CIG are the BCPs. The Boolean variable breakForFlag is used to determine whether obtaining a BCP, if yes, set up breakForFlag = true, do not search the next edge of the destination node, otherwise, set up breakForFlag = false, continue to search the next edge. Finally, output basis component path set (lines 35–36), where line 35 is used to replace the number of basis component path set with the component name and connector name.

We present the BCPA algorithm of our proposed technique to generate the basis component path set as illustrated in Algorithm 2.

3.3 Illustration of Working of BCPA Algorithm

We explain the working of BCPA algorithm by using the example from component C_1 to component C_7. First, we replace the component name and connector name with number of the CIG as shown in Table 1.

Node Name	Number
C_1	1
C_2	2
C_3	3
C_4	4
C_5	5
C_6	6

Node Name	Number
C_7	7
B_1	8
B_2	9
B_3	10
B_4	11

Table 1. Number of component and connector

First $N \leftarrow 1$, $C_i \leftarrow 1$, $C_j \leftarrow 7$, EG $\leftarrow$ CIG. And then add C_i (number = 1) to the set of current path BCP[1], so, BCP[1] = $\{1\}$. Search the next node C_k of C_i (number = 1) in CIG. Because C_k (number = 8) exists, then add C_k (number = 8) to the set BCP[1], so, BCP[1] = $\{1, 8\}$. Determine the last node of BCP[1] is 8, not the destination node C_j (number = 7), then $C_i \leftarrow 8$. Search the next node C_k of C_i (number = 8) in CIG. Because C_k (number = 3, 9, 10) exists, then add C_k (number = 3) to BCP[1], so, BCP[1] = $\{1, 8, 3\}$. Determine the last node of BCP[1] is 3, not the destination node C_j (number = 7), then $C_i \leftarrow 3$. Search the next node C_k of C_i (number = 3) in CIG. Because C_k (number = 10) exists, then add C_k (number = 10) to BCP[1], so, BCP[1] = $\{1, 8, 3, 10\}$. Determine the last node of BCP[1] is 10, not the destination node C_j (number = 7), then $C_i \leftarrow 10$. Search the next node C_k of C_i (number = 10) in CIG. Because C_k (number = 4, 5) exists, then add C_k (number = 4) to BCP[1], so, BCP[1] = $\{1, 8, 3, 10, 4\}$. Determine the last node of BCP[1] is 4, not the destination node C_j (number = 7), then $C_i \leftarrow 4$. Search the next node C_k of C_i (number = 4) in CIG. Because C_k (number = 11) exists, then add C_k (number = 11) to BCP[1], so, BCP[1] = $\{1, 8, 3, 10, 4, 11\}$. Determine the last node of BCP[1] is 11, not the destination node C_j (number = 7), then $C_i \leftarrow 11$.

Algorithm 2 BCPA

Input: CIG
Output: basis component path set
begin
BCPS = ∅;
foreach component C_i in CIG **do**
  foreach component C_j in CIG **do**
    EG ← CIG;
    $N \leftarrow 1$
    add C_i to the set of current path BCP[N];
    foreach component C_k in CIG **do**
      breakForFlag ← false;
      if (edge (C_i, C_k) exists in CIG) **then**
        add C_k to the current path BCP[N];
        if (C_k is the destination node C_j) **then**
          foreach component C_m in EG **do**
            foreach component C_n in EG **do**
              if (edge (C_m, C_n) exists in EG) **then**
                if (edge (C_m, C_n) exists in BCP[N]) **then**
                  add the current path BCP[N] to the set of BCPS;
                  delete the edges of the EG corresponding to the edges in BCP[N];
                  $N \leftarrow N + 1$;
                  BCP[N] ← BCP[$N - 1$];
                  breakForFlag ← true;
                end if
              end if
            end for
            if (breakForFlag == true) **then**
              break;
            end if
          end for
        else
          $C_i \leftarrow C_k$;
        end if
      end if
    end for
  end for
end for
replace the number of BCPS with the component name and the connector name;
output BCPS;
end

Search the next node C_k of C_i (number = 11) in CIG. Because C_k (number = 7) exists, then add C_k (number = 7) to BCP[1], so, BCP[1] = $\{1, 8, 3, 10, 4, 11, 7\}$. Determine the last node of BCP[1] = $\{1, 8, 3, 10, 4, 11, 7\}$ is 7, that is the destination node C_j (number = 7). In addition, there is an edge from EG which existed in the BCP[1], so, BCP[1] = $\{1, 8, 3, 10, 4, 11, 7\}$ is a BCP from node 1 to node 7. Add the BCP[1] = $\{1, 8, 3, 10, 4, 11, 7\}$ to the set of BCPs.

After that delete the edges of the EG corresponding to the edges of BCP[1] and delete the last node (destination node) of BCP[1]. Calculate the number of next BCP after $N \leftarrow N + 1$ and obtain BCP[N] after BCP[N] $\leftarrow$ BCP[$N - 1$], so, BCP[2] = $\{1, 8, 3, 10, 4, 11\}$, then $C_i \leftarrow 11$ (the last node of BCP[2]). Continue to search the other next node C_k of C_i (number = 11) in CIG. Because other C_k does not exist, then delete the last node of BCP[2], so, BCP[2] = $\{1, 8, 3, 10, 4\}$, then $C_i \leftarrow 4$ (the last node of BCP[2]). The algorithm will repeat the steps to get other two independent BCPs from node 1 to node 7 as shown as follows:

$$\begin{aligned} \text{BCP}[2] &= \{1, 8, 3, 10, 5, 11, 7\} \\ \text{BCP}[3] &= \{1, 8, 10, 4, 11, 7\} \end{aligned}$$

Finally, convert each number of the BCP[1], BCP[2], and BCP[3] into its corresponding component name and connector name, thus, obtain the basis component path set from component C_1 to C_7 is shown as follows:

$$\begin{aligned} \text{BCPS} = \{&C_1 \rightarrow B_1 \rightarrow C_3 \rightarrow B_3 \rightarrow C_4 \rightarrow B_4 \rightarrow C_7, \\ &C_1 \rightarrow B_1 \rightarrow C_3 \rightarrow B_3 \rightarrow C_5 \rightarrow B_4 \rightarrow C_7, \\ &C_1 \rightarrow B_1 \rightarrow B_3 \rightarrow C_4 \rightarrow B_4 \rightarrow C_7\}. \end{aligned}$$

By calculating according to Equation (1), we get the number of BCPs which is: $|\text{BCP}_{C_1}^{C_7}| = (4 + 4 + 1) - (5 + 3) + 2 = 3$. So, the BCPA algorithm can generate the basis component path set from component C_1 to component C_7.

4 CASE STUDY

We developed a tool based on C2-style architecture to generate BCPs. In this section, we use an example to show performance of our method.

4.1 KLAX System

To validate the proposed method, we choose KLAX [7] system as a case study to generate BCPs. The CIG is shown in the Figure 3, it illustrates basis interaction between 16 components *GraphicsBinding* and *LayoutManager*, etc. and 6 connectors *ALAConn* and *LTConn*, etc. where are 72 interfaces, for example, *GraphBinding*.I_{pt_o} is an interface that represents the top output of the component *GraphsBinding*, *ALAC*.I_{nb_i} is an interface that represents the bottom input of the connector *ALAConn*.

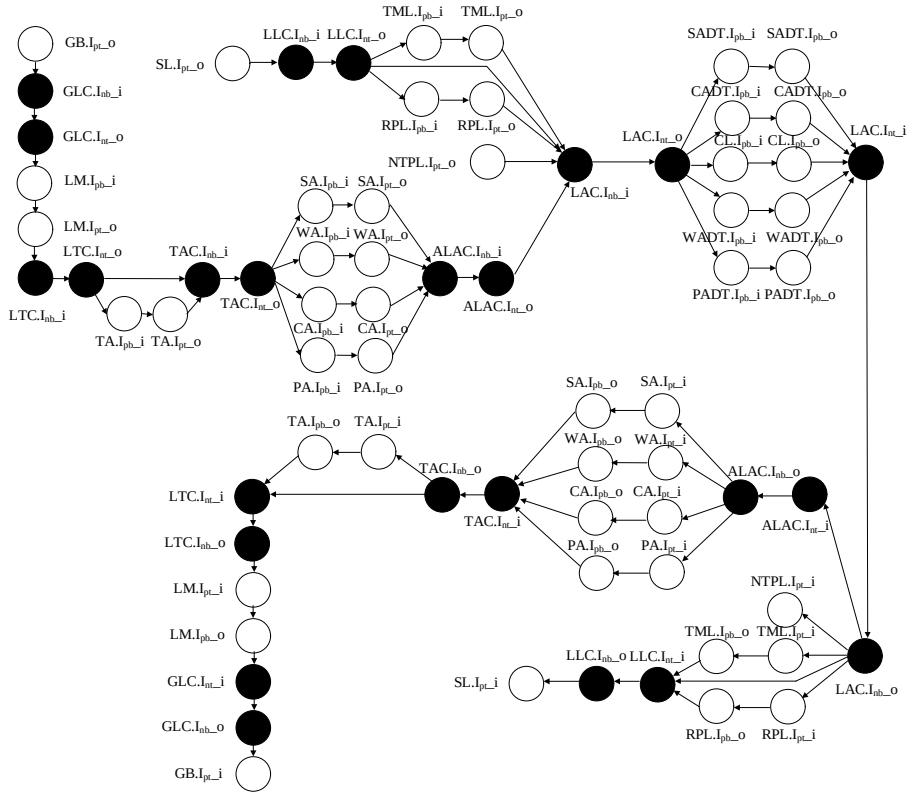

Figure 3. CIG of KLAX system

4.2 Experiment Results

Results of the BCPs are listed for KLAX system in Table 2, where the first column represents component. The second column represents the number of BCPs from the component to another component. In the case of component GraphicsBinding, the last 2 in |BCP| represents that there exists two BCPs from GraphicsBinding to PaletteArtist, the first 5 in |BCP| represents that there exists five BCPs from GraphicsBinding to ClockLogic, and 0 represents that there does not exist any BCP between components.

From Table 2, the less 0 in |BCP|, the more basis component path exists between components, the more component path exists between components. We can obtain the basis component path set from a component to another component according to the BCPA algorithm, and the number of BCPs is less than or equal to the number of component paths. So, the test scale is significantly reduced after applying basis component path coverage criteria. This method can be achieved

Component	|BCP|
GraphicsBinding	1, 1, 2, 2, 2, 2, 0, 0, 0, 0, 5, 5, 5, 5, 5
LayoutManager	1, 1, 2, 2, 2, 2, 0, 0, 0, 0, 5, 5, 5, 5, 5
TileArtist	1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 4, 4, 4, 4, 4
StatusArtist	2, 2, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1
ChuteArtist	2, 2, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1
WellArtist	2, 2, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1
PaletteArtist	2, 2, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1
StatusLogic	0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 3, 3, 3, 3, 3
NextTilePlacingLogic	0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1
TileMatchLogic	0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1
RelativePosLogic	0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1
ClockLogic	5, 5, 4, 1, 1, 1, 1, 3, 1, 1, 1, 0, 0, 0, 0
StatusADT	5, 5, 4, 1, 1, 1, 1, 3, 1, 1, 1, 0, 0, 0, 0
ChuteADT	5, 5, 4, 1, 1, 1, 1, 3, 1, 1, 1, 0, 0, 0, 0
WellADT	5, 5, 4, 1, 1, 1, 1, 3, 1, 1, 1, 0, 0, 0, 0
PaletteADT	5, 5, 4, 1, 1, 1, 1, 3, 1, 1, 1, 0, 0, 0, 0

Table 2. Results of number of BCPs for KLAX system

with less test suites to find more architecture specification errors. Testers can cover all of the component paths in software architecture with smaller number of test suites.

5 RELATED WORK

The paths that will be tested must be generated or determined before path testing. Some kinds of path generating methods and software architecture coverage methods have been discussed in this section.

Bertolino and Marré provided a path generation method [9] by using a reduced Control Flow Graph (CFG). Although all the statements and branches can be covered by the set of paths, it cannot be assured that the set of paths generated by this method is a basis set of paths. Poole discussed a basis set of paths generation method [10] on the depth first search in CFG. It uses a recursive search in the CFG. As Poole's method, the loop is not taken into account. In addition, this method did not consider how to choose the successor of multiple-successors node in a CFG to build a basis path during the construction of the basis set of paths.

A testing mechanism proposed by McCabe [8], aim is to derive the cyclomatic complexity measure of a procedural design and use this as a guide for defining a basic set of execution paths. He also proved that the cyclomatic complexity of a section of source code is the count of the number of linearly independent paths through the source code. The baseline method is different from the basis path testing, it is a technique to derive a set of basis paths through the CFG generated from the tested program, and it is equal to the cyclomatic complexity. The idea is to start with

a baseline path, then very exactly one decision outcome to generate each successive path until all decision outcomes have been varied, at which time a basis path would have been generated [11].

Zhang and Mei analyzed dependence relationship of the predicate [12]; pick the shortest path as the baseline path instead of the longest path which has much branch nodes. In this way, using "baseline path+flip" to generate set of independent paths, may avoid selecting infeasible paths. Yan and Zhang presented a method for generating a finite set F of feasible paths which satisfies the basis path coverage criterion [13]. Then, they found a minimal subset S of set F such that S satisfies the test coverage criterion. The first step should check the feasibility of all paths and feasibility checking is quite time-consuming.

Du and Dong used cyclomatic complexity in generating a set of linearly independent paths [14]. Many basis paths are infeasible because of data dependences exist in variables involved in decision node. They combine the baseline method with the dependence relationship to avoid selecting infeasible paths. These methods did not handle loops.

Rosenblum defined two formal adequate test models for component-based software [15]. The first model is known as C-adequate-for-$\mathcal{P}$, which is defined for adequate unit testing of a component where C refers to test adequacy criteria and $\mathcal{P}$ refers to a program. The other model is known as C-adequate-on-$\mathcal{M}$, which is defined for adequate integration testing of component based system. Both test models are defined based on subdomain-based test adequacy criteria defined by Frankl and Weyuker [16].

Jin and Offutt proposed a technology of generating test cases [17] in view of architecture description language Wright, according to Interface Connectivity Graph (ICG) and Behavior Graph (BG). And developed testing criteria for generating software architecture level tests from software architecture descriptions.

Gao et al. focused on component test coverage issues, and proposed test models (CFAGs and D-CFAGs) [18] to represent a component's API-based function access patterns in static and dynamic views. A set of component API-based test criteria is defined based on the test models, and a dynamic test coverage analysis approach is provided.

Hashim et al. presented Connector-based Integration Testing for Component-based Systems (CITECB) with an architecture test coverage criteria [19], and describe the test models used that are based on probabilistic deterministic finite automata which are used to represent gate usage profiles at run-time and test execution. It also provides a measuring mechanism of how well the existing test suite are covering the component interactions and provides a test suite coverage monitoring mechanism to reveal the test elements that are not yet covered by the test suites. The model extraction technique used to generate the CITECB test models is a simple and less time consuming process. In addition to that, these test models are able to closely represent the component interactions as they are extracted directly from the system.

Muccini et al. proposed a specialization and refinement of general approach for software architecture based conformance testing [20], he deals with the software architecture to code mapping rules imposed by the C2 framework helps to limit the mapping problem, and allows a systematic testing approach.

Lun et al. presented a dependency edge coverage method [21] for software architecture. We described three types of dependency edge, named dependency edge from component to connector, dependency edge from connector to component, and dependency edge from connector to connector. We used the dependency coverage of component to connector, the dependency coverage of connector to component, the dependency coverage of connector to connector, and the dependency coverage of C2-style architecture metrics standard to evaluate the effectiveness of dependency edge coverage criteria.

6 CONCLUSION

We analyzed the problem of BCP for C2-style architecture. To develop the test model, CIG is constructed from components, connectors, interfaces, and relationships between components and connectors abstracted the behavior of interactive between components and connectors for C2-style architecture, and BCP is formalized based on the CIG. We propose an automatic method to generate BCPs. For verifying the method, the KLAX system is illustrated as an example. The experiment results reveal that the C2-style architecture realizable the number of BCPs $|\mathrm{BCP}_{V_i}^{V_j}|$ accordance with modified the cyclomatic complexity $V(G)$. The BCP on C2-style architecture testing can be useful to predict how much effort should be expected to be necessary for testing given a C2-style architecture.

Much work also remains to be done to validate the practical results. It is our plan to apply model in new cases and in new domains. In addition, we need to consider the impact with respect to the C2-style architecture quality when we introduce measurement mechanism and technique. Thus, another research area that is of interest is to investigate the impact of the choice of measurement mechanisms for analysis.

Acknowledgement

The authors are grateful to the anonymous referees for their detailed comments and insightful suggestions, which helped in refining and improving the presentation of the paper. 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. 12541250.

REFERENCES

[1] Perry, D. E.—Wolf, A. L.: Foundations for the Study of Software Architecture. ACM SIGSOFT Software Engineering Notes, Vol. 17, 1992, No. 4, pp. 40–52, doi: 10.1145/141874.141884.

[2] Malavolta, I.—Lago, P.—Muccini, H.—Pelliccione, P.: What Industry Needs from Architectural Languages: A Survey. IEEE Transactions on Software Engineering, Vol. 39, 2013, No. 6, pp. 869–891, doi: 10.1109/tse.2012.74.

[3] Muccini, H.—Bertolino, A.—Inverardi, P.: Using Software Architecture for Code Testing. IEEE Transactions on Software Engineering, Vol. 30, 2004, No. 3, pp. 160–171, doi: 10.1109/tse.2004.1271170.

[4] Chen, J. F.—Lu, Y. S.—Wang, H. H.: Component Security Testing Approach Based on Extended Chemical Machine. International Journal of Software Engineering and Knowledge Engineering, Vol. 22, 2012, No. 1, pp. 59–83, doi: 10.1142/s0218194012500039.

[5] Lun, L. J.—Chi, X.—Ding, X. M.: Edge Coverage Analysis for Software Architecture Testing. Journal of Software, Vol. 7, 2012, No. 5, pp. 1121–1128, doi: 10.4304/jsw.7.5.1121-1128.

[6] Bertolino, A.—Inverardi, P.—Muccini, H.: An Explorative Journey from Architectural Tests Definition down to Code Test Execution. Proceedings of the 23rd International Conference on Software Engineering, May 2001, pp. 211–220.

[7] Taylor, R. N.—Medvidovic, N.—Anderson, K. M.—Whitehead, E. J.—Robbins, J. E.: A Component- and Message-Based Architecture Style for GUI Software. IEEE Transactions on Software Engineering, Vol. 22, 1996, No. 6, pp. 390–406.

[8] McCabe, T. J.: A Complexity Measure. IEEE Transactions on Software Engineering, Vol. 2, 1976, No. 4, pp. 308–320, doi: 10.1109/tse.1976.233837.

[9] Bertolino, A.—Marré, M.: Automatic Generation of Path Covers Based on the Control Flow Analysis of Computer Programs. IEEE Transactions on Software Engineering, Vol. 20, 1994, No. 12, pp. 885–899, doi: 10.1109/32.368137.

[10] Poole, J.: A Method to Determine a Basis Set of Paths to Perform Program Testing. Available on: `http://hissa.nist.gov/publications/nistir5737`, 1995.

[11] Watson, A. H.—McCabe, T. J.: Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric. Technical Report NIST Special Publication 1996.

[12] Zhang, Z. L.—Mei, L. X.: An Improved Method of Acquiring Basis Path for Software Testing. Proceedings of 5th International Conference on Computer Science and Education, August 2010 pp. 1891–1894.

[13] Yan, J.—Zhang, J.: An Efficient Method to Generate Feasible Paths for Basis Path Testing. Information Processing Letters, Vol. 107, 2008, No. 3, pp. 87–92.

[14] Du, Q. F.—Dong, X.: An Improved Algorithm for Basis Path Testing. International Conference on Business Management and Electronic Information, May 2011, pp. 175–178.

[15] Rosenblum, D. S.: Adequate Testing of Component-Based Software. Technical Report TR97-34, 1997.

[16] Frankl, P. G.—Weyuker, E. J.: An Applicable Family of Data Flow Testing Criteria. IEEE Transactions on Software Engineering, Vol. 14, 1988, No. 10, pp. 1483–1498, doi: 10.1109/32.6194.

[17] Jin, Z. Y.—Offutt, J.: Deriving Tests from Software Architectures. Proceedings 12th International Symposium on Software Reliability Engineering, November 2001, pp. 308–313.

[18] Gao, J.—Espinoza, R.—He, J.: Testing Coverage Analysis for Software Component Validation. 29th Annual International Computer Software and Applications Conference, July 2005, pp. 463–470.

[19] Hashim, N. L.—Ramakrishnan, S.—Schmidt, H. W.: Architectural Test Coverage for Component-Based Integration Testing. Seventh International Conference on Quality Software, October 2007, pp. 262–267, doi: 10.1109/qsic.2007.4385505.

[20] Muccini, H.—Dias, M.—Richardson, D. J.: Systematic Testing of Software Architectures in the C2 Style. Lecture Notes in Computer Science, Vol. 2984, 2004, pp. 295–309, doi: 10.1007/978-3-540-24721-0_22.

[21] Lun, L. J.—Chi, X.—Ding, X. M.: C2-Style Architecture Testing and Metrics Using Dependency Analysis. Journal of Software, Vol. 8, 2013, No. 2, pp. 276–285.

Lijun Lun is Professor of computer science and information engineering at Harbin Normal University of Harbin. He received his B.Sc. and Master's degree in computer science and technology from Harbin Institute Technology of Computer Science and Technology, China, in 1986 and 2000, respectively. He has published more than 70 papers in international conferences and journals. Currently, he teaches and conducts research in the areas of software architecture, software testing and software metrics, etc.

Shaoting Wang received her B.Sc. in computer science and technology from Harbin Normal University of Computer Science and Information Engineering, China, in 2012. Now she is studying for her Master's degree in Harbin Normal University of Computer Science and Information Engineering, China. Currently, her research interest includes software architecture testing.

Xin Chi received her B.Sc. in computer science and technology from Harbin Normal University of Computer Science and Information Engineering, China, in 2013. She has published more than 10 papers in international conferences and journals. Currently, her research interests include software architecture testing and software metrics, etc.

Hui Xu received her Master's degree in computer science and technology from Harbin Normal University, China, in 2009. Now she is studying for her Ph.D. in Harbin Engineering University of Computer Science and Technology, China. She has published more than 10 papers in international conferences and journals. Currently, her research interests include social networks and social computing.