Coverage Criteria for Component Path-oriented in Software Architecture

Lijun Lun, Xin Chi, Hui Xu

Abstract—**Software architecture testing is an important method of assuring quality and enhancing reliability and component path coverage is an essential criterion for software architecture testing adequacy. Since the rigid component path cover testing is infeasible, we select and test some key component paths, which outweigh others on affecting the overall quality of the system, to enhance software architecture test efficiency and profit. This paper presents a set of component path coverage criteria for C2-style architecture, and proposes two component path coverage criteria, component path with node-sequence coverage criterion and component path with edge-sequence coverage criterion, and proves the subsumption relationships among them. We propose two algorithms to calculate the component path coverage rate on these two component path coverage criteria. Experimental results show that, for top and bottom components, the component path of length N coverage rate decreases from 57.14% to 34.78%, the component path with node-sequence of node N coverage rate increases from 64.45% to 100%, and the component path with edge-sequence of edge N coverage rate increases from 66.26% to 100%. However for middle levels components, the component path of length N coverage rate decreases from 26.92% to 8.70%, the component path with node-sequence of node N coverage rate decreases from 7.80% to 0.16%, and the component path with edge-sequence of edge N coverage rate decreases from 7.64% to 0.08%. Experimental result shows that the proposed component path coverage criteria provide a good foundation for software architecture testing practice and the further research.**

Index Terms—**software architecture, C2-style, component path, coverage criteria.**

I. Introduction

SOFTWARE architecture is the product of the first stage in the software development process [1], which is represented the early design decisions of the software system. It determines the compositions of the system at the same time, but also restricts the interaction relationships between the compositions. If the faults of the architecture layer are propagated to the follow stages, the cost of maintenance will be greatly increased. Therefore, it is very important to test software architecture as early as possible.

There are four phases in software testing which are static analysis, selection of test coverage criteria, generation of test suites, execution of test suites and analysis of the reliability of the execution results [2]. There are differences and connections between software architecture testing and traditional software testing. The purpose of the software architecture testing is to find the architecture design faults [3], which generates the test plan and test suites to guide code testing, which is very different from the traditional software testing. The test plan and test suites of software architecture will be refined and verified by code layer testing, which makes the software architecture test closely related to traditional software testing.

Software architecture testing technology divided into includes two categories [4], one is software architecture analysis, the other is software architecture testing. Software architecture testing are mainly two types, one is the test software architecture, that is using simulation methods to test the interface behaviors of software architecture, or test the interaction relationships between components, or test the communication relationships between components, so as to analysis the difference between the behavior and target system. The other is to test and guide object code generation based on software architecture. These two kinds of software architecture testing involve the core technology of component interaction. In software architecture, component interaction is an important part in software architecture testing [5]. An appropriate component interaction for software architecture testing can reduce test cost. Different component interaction, need different test cost. So, to reduce the test cost is an important target to determine the component interaction as soon as possible.

Component interaction is a use relationship, which represents a component specification changes may affect the change used by other components. In software architecture, components communicate with each other to share information, and to provide a system function [6]. Components need to conform a component model so as to allow them to be independently deployed and composed as is i.e. achieve the purpose of their creation [7]. Software architecture functions provided by multiple components, thus modifying a component may affect the function of the whole system [8]. Component interaction should satisfy a certain coverage criterion to achieve component function.

We have presented a set of component path coverage criteria for C2-style architecture [6], these coverage criteria provide a coverage measure to quantify the testing activity and thus contribute to the improvement of the quality of this activity in C2-style architecture. This paper proposes two component path coverage criteria, component path with node-sequence coverage criterion and component path with edge-sequence coverage criterion. Firstly, set of interactions relationships is defined corresponding to the relationship between component and connector. Then the component interaction graph of C2-style architecture is constructed on the basis of these interaction relationships. Based on the component interaction graph introduced, component path

Manuscript received November 1, 2017; revised April 3, 2018. This work was supported in part by the Natural Science Foundation of Heilongjiang Province of China under Grant F201036 and by the Scientific Research Foundation of Heilongjiang Provincial Education Department of China under Grant 12541250.

Lijun Lun is with the College of Computer Science and Information Engineering, Harbin Normal University, Harbin, 150080, China, e-mail: lunlijun@yeah.net.

Xin Chi is with the Harbin Normal University, China, e-mail: xinc1990@163.com.

Hui Xu is with the Heilongjiang University of Chinese Medicine, China, e-mail: xuhui8413@163.com.

with node-sequence coverage criterion and component path with edge-sequence coverage criterion are proposed, the subsumption relationships among them are proven, two algorithms to compute the component path with node-sequence and with edge-sequence coverage rate on component path coverage criteria are proposed, some experimental results of these component path coverage criteria are discussed. And finally conclusion and future work are given.

II. Related Work

We have proposed component path coverage criteria to achieve component interactions, which are a significant challenge for software architecture because systematically executing all component paths in software architecture are costly. This section reviews closely related work on this topic. A number of coverage criteria have been proposed in the literature to handle the problem from different aspects: path coverage criteria and software architecture coverage criteria.

A. Path Coverage Criteria

Paige proposed level-i path coverage criterion [9]. Informally, the criterion starts with testing all elementary paths from the begin node to the end node. Then, if there is an elementary subpath or cycle that has not been exercised, the subpath is required to be checked at the next level.

McCabe proposed basis path coverage criterion [10]. In a real software system, even if a program is not so complex, its number of paths may be enormous, so, the path coverage of the program requires enormous resources. The basic path coverage is to find a path set of reducing the scale of path coverage by means of program graph, makes each path of path set corresponds to a test suite, so as to achieve basis path coverage.

Miller proposed DD-PATH coverage criterion [11]. The DD-PATH is a chain from a decision node to another decision node in program graph. The definitions of different type of DD-PATH that a chain can be reduced to are given follows: (1) A single node with an in-degree = 0, (2) A single node with an out-degree = 0, (3) A single node with an in-degree $>=$ 2 or out-degree $>=$ 2, (4) A single node with an in-degree = 1 and out-degree = 1, and (5) The chain is of a maximal length $\geq$ 1.

Gourlay proposed length-n path coverage criterion ($PATH_n$) [12]. The length-n path is consists of consecutive n statement in the program P. The length-n path coverage criterion denoted the set of all paths in program of length less than or equal to n. $PATH_1(P)$ is the set of all nodes of program P. $PATH_2(P)$ is the set of all nodes together with the set of all branches of program P.

Li et al. proposed Length_N path coverage criterion [13]. The automated approach for generating test data was proposed by solving multi-object function. An efficient approach to automated generation of structural test data is to breed search iteratively by profiling of program execution. The structured test suite is generated by the way of path coverage. Through different path, it can generate different groups of test suite which can coverage different path. Researchers proposed a number of improved path coverage criteria.

Li et al. proposed a uniformed path coverage criterion, namely Length-n Subpath Coverage Criterion (LSC(n)) [14]. The Length-n Subpath is consists of consecutive n branches in control flow, it is a subset of complete path. LSC(n) measures the covering situation of Length-n Subpath of program, a consecutive sub-sequence from a complete path with n branches in control flow. By adjusting the n value, it can obtain different program spectrum and the program behavior reflected. With the increase of n, the strength of LSC(n) increases monotonously.

B. Software Architecture Coverage Criteria

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. In essence, software architecture coverage is a kind of coverage based on software architecture specification. The adequacy testing of two models is based on the test adequacy condition of subdomains.

Stafford et al. described chaining which the goal is to reduce the portions of an architecture that must be examined by an architect for some purpose, such as testing or debugging [16]. In chaining, links represent the dependence relationships that exist in an architectural specification. Links connect elements of the specification that are directly related, producing a chain of dependencies that can be followed during the analysis.

Richardson et al. suggested the family of architecture-based test criteria based on the CHAM model [17], such as all-data-elements criteria, all-processing-elements criteria, all-connecting-elements criteria, all-transformations criteria, all-transformation-system criteria, and all-data-dependences criteria.

Jin and Offitt [18] defined six architecture relationships between software architecture units, these relationships are key factors in software architecture description. These relationships can also be used to define the software architecture testing path, so as to further define the test criteria of software architecture level. The six software architecture relationships include component (connector) internal transfer relation, component (connector) internal sequencing relation, component (connector) internal sequencing relation, component (connector) internal relation, N_C relation or C_N relation, direct component relation, and indirect component relation. According to these six relationships, they defined five software architecture testing criteria to cover all identified software architecture relationships. These coverage criteria can be epitomized as individual component interface coverage criterion, individual connector interface coverage criterion, all direct component-to-component coverage criterion, all indirect component-to-component coverage criterion, and all connected components coverage criterion.

Hashim et al. presented Connector-based Integration Testing for Component-based Systems (CITECB) with an architectural 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 suites 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.

Lun and Chi presented a component dependency path coverage approach based on component dependency graph, and proposed three component dependency path coverage criteria [6], which are direct component dependency path coverage criterion, indirect component dependency path coverage criterion, and Length-N component dependency path coverage criterion. It covered all testing component and connector, and reduced scale of testing coverage set. Meanwhile, they presented three algorithms to compute the component dependency path coverage rate on these component dependency path coverage criteria.

III. C2-Style Architecture Model

This section first introduces the related concepts of the C2-style architecture, then gives the definition of the component path.

A. C2-Style Architecture Representation

We have selected the C2-style architecture as a vehicle for exploring our ideas because it provides a number of useful rules for high-level system composition, demonstrated in numerous applications across several domains [20]; at the same time, the rules of the C2-style are broad enough to render it widely applicable [3].

The C2-style architecture [21] consists of components, connectors, and their constraints. All components and connectors have two interfaces, "top" and "bottom". The top (bottom) of a component can only be attached to the bottom (top) of one connector. It is not possible for components to be attached directly to each other. Each connector always has to act as intermediaries between them. Furthermore, a component cannot be attached to itself. However, connector can be attached together. In this case, each connector considers the other as a component with regard to the publication and forwarding of events. Component communicates by exchanging two types of events: service requests to top of the component and notifications of completed services to bottom of the component.

We have defined our intermediate representation Component Interaction Graph (CIG) model [22] and illustrate how a C2-style architecture can be represented using our notation. CIG is used to depict the interaction relationships between interface of component and interface of connector.

Definition 1 Let CIG = (V, E, V_{start}, V_{end}) be a component interaction graph, where V = Comp $\cup$ Conn is the set of nodes, Comp is a finite set of components, each component $Comp_i \in$ Comp has four interfaces, they are top output interface $Comp_i.I_{pt}\_o$, top input interface $Comp_i.I_{pt}\_i$, bottom output interface $Comp_i.I_{pb}\_o$, and bottom input interface $Comp_i.I_{pb}\_i$. Conn is a finite set of connectors, each connector $Conn_j \in$ Conn has four interfaces too, they are top output interface $Conn_j.I_{nt}\_o$, top input interface $Conn_j.I_{nt}\_i$, bottom output interface $Conn_i.I_{nb}\_o$, and bottom input interface $Conn_i.I_{nb}\_i$. E = $e_{Comp-Conn} \cup e_{Conn-Comp} \cup e_{Conn-Conn}$ is a finite set of edges, where $e_{Comp-Conn}$ = $\{e \mid e \in (Comp_i.I_{pt}\_o, Conn_j.I_{nb}\_i) \vee (Comp_i.I_{pb}\_o, Conn_j.I_{nt}\_i)\}$ represents the set of edges from top (bottom) output interface of component $Comp_i$ to the bottom (top) input interface of connector $Conn_j$. $e_{Conn-Comp}$ = $\{e \mid e \in (Conn_i.I_{nt}\_o, Comp_j.I_{pb}\_i) \vee (Conn_i.I_{nb}\_o, Comp_j.I_{pt}\_i)\}$ represents the set of edges from the top (bottom) output interface of connector $Conn_i$ to the bottom (top) input interface of component $Comp_j$. $e_{Conn-Conn}$ = $\{e \mid e \in (Conn_i.I_{nt}\_o, Conn_j.I_{nb}\_i) \vee (Conn_i.I_{nb}\_o, Conn_j.I_{nt}\_i)\}$ represents the set of edges from the top (bottom) output interface of connector $Conn_i$ to the bottom (top) input interface of connector $Conn_j$. $V_{start} \subseteq$ Comp is the set of initial component nodes, these components transmit messages only. That is V_{start} = $\{Comp_i \mid Comp_i.I_{pb}\_i = \emptyset \wedge Comp_i.I_{pb}\_o = \emptyset, Comp_i \in$ Comp$\}$. $V_{end} \subseteq$ Comp is the set of terminal component nodes, these components receive messages only. That is V_{end} = $\{Comp_i \mid Comp_i.I_{pt}\_o = \emptyset \wedge Comp_i.I_{pt}\_i = \emptyset, Comp_i \in$ Comp$\}$.

In C2-style architecture, a component (connector) can interact with the other component (connector) in several ways, i.e., from component to connector, from connector to component, and from connector to connector. The CIG for C2-style architecture should be able to represent these interactions between components and connectors.

In order to construct a representation for the CIG, we carry out static analysis of the C2-style specification. First, we identify all components and connectors and represent as nodes, then identify all interaction relationships between components and connectors and represent as edges. If there exists a information flow from component $Comp_i$ to connector $Conn_j$, in such a case, an edge $e \in e_{Comp-Conn}$ is added to connect from the top (bottom) output interface of $Comp_i$ to the bottom (top) input interface of $Conn_j$ of CIG. If there exists a information flow from connector $Conn_i$ to component $Comp_j$, an edge $e \in e_{Conn-Comp}$ is added to connect from the top (bottom) output interface of $Conn_i$ to the bottom (top) input interface of $Comp_j$ of CIG. If there exists a information flow from connector $Conn_i$ to connector $Conn_j$, an edge $e \in e_{Conn-Conn}$ is added to connect from the top (bottom) output interface of $Conn_i$ to the bottom (top) input interface of $Conn_j$ of CIG.

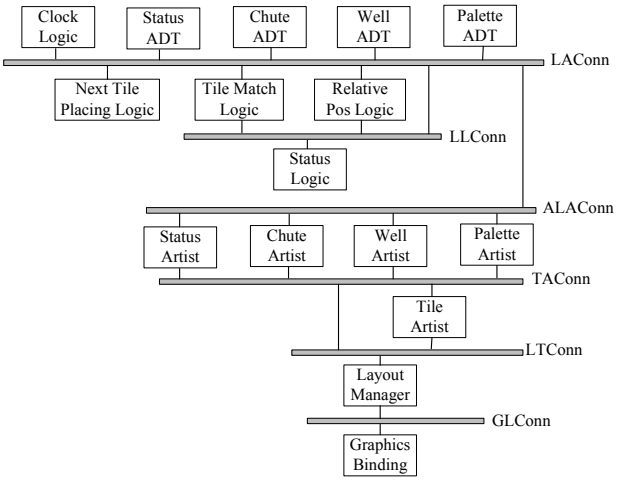

Fig. 1. KLAX Architecture in the C2-Style

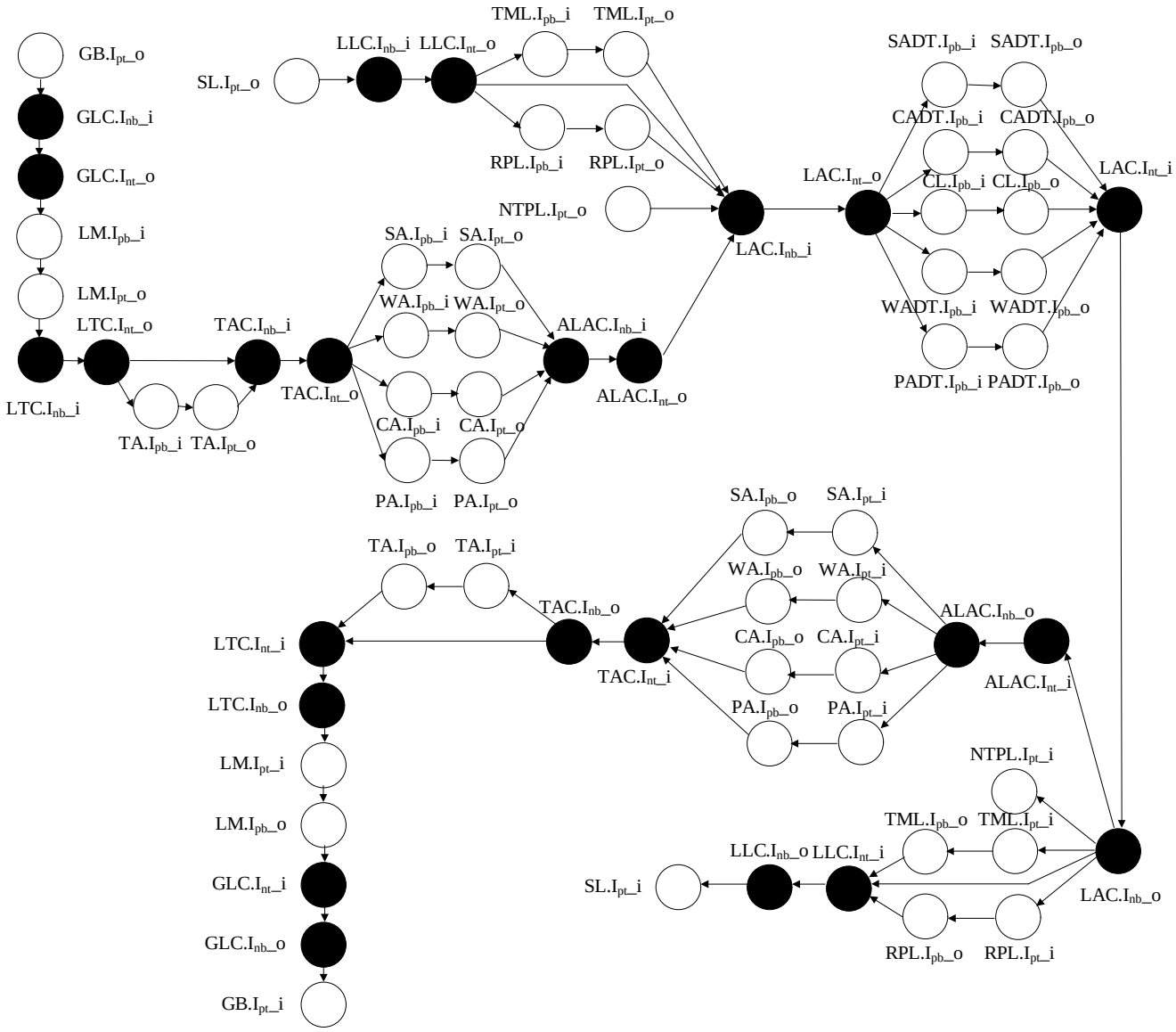

Fig. 2. CIG of KLAX System

In order to illustrate our approach in a better way, we use an example KLAX video game application to illustrate our proposed notions [20]. KLAX system includes 16 components and 6 connectors, which is depicted in Fig. 1. Where the rectangle node represents component, such as GraphicsBinding and TileArtist etc. The long rectangle with shadow node represents connector, such as LAConn and TAConn etc. The edge between component and connector, and between connectors represents that there exists messages transmission between component and connector, such as the edge between GraphicsBinding and GLConn represents that there exists messages transmission between GraphicsBinding and GLConn, and the edge between LTConn and TAConn represents that there exists messages transmission between LTConn and TAConn.

According to the construction method of CIG [22], Fig. 2 shows the corresponding CIG for the example KLAX system of Fig. 1 according to C2-style architecture specification [21]. In order to simplify the representation, the name of the component and the connector are abbreviated. Where nodes represent the interface of the component and the connector, and component interface with a hollow circle, connector interface with a solid circle represents. $GB.I_{pt\_}o$, $SL.I_{pt\_}o$, and $NTPL.I_{pt\_}o$ are initial nodes. $CL.I_{pb\_}i$, $PADT.I_{pb\_}i$ and so on are terminal nodes.

B. Component Path

Software architecture describes the components, connectors, and their relationships in the system, all these relationships have an impact on interactions between component and connector. Interaction relationships between component and connector can be represented as component path.

Definition 2 Let $CIG =< V, E, V_{start}, V_{end} >$ be a component interaction graph for C2-style architecture, C_s, C_{s+1}, ..., $C_t \in$ V. A path is a sequence nodes $C_s \rightarrow C_{s+1} \rightarrow \ldots \rightarrow C_t$ such that $(C_i, C_j) \in$ E for i = 2, 3, ..., t-2, j = i+1, i+2, ..., t-1, denoted as π_P, The length of π_P is the number of edges that it crosses. If $C_s \in$ Comp $\wedge$ $C_t \in$ Comp, the path π_P is called component path, denoted as π_{CP}.

From the definition 2, we can see that the π_{CP} has two forms according to the type of edges, one is all edges from

the beginning of top output interface of component and connector to the end of bottom input interface of component and connector, other is all edges from the beginning of bottom output interface of component and connector to the end of top input interface of component and connector.

For example in Fig. 2, because (LayoutManager, GLConn) ∈ E, (GLConn, GraphicsBinding) ∈ E, thus, LayoutManager → GLConn → GraphicsBinding is a π_{CP} of length is 2. Similarly, LayoutManager → GLConn → TileArtist → TAConn → StatusArtist → ALAConn → LAConn → ClockLogic is a π_{CP} of length is 7.

IV. Component Path Coverage Criteria

Coverage criteria are key problem in software testing, how do we decide if a test set is adequate? This question was first addressed by Goodenough and Gerhart when they considered the idea of a test adequacy criterion [23], that is, coverage criteria that defines what makes an adequate test, they can also be used as a measurement of test quality. Component path coverage criteria are one or more rules applied to test suite. The component path coverage criteria guideline allow less testing blindness, guarantee testing adequacy, help testers to develop test strategies, generate test suites, detect as many faults as possible, and decide when to stop software architecture testing. In this section, we propose a set of component path coverage criteria of software architecture using the CIG. The number of test suites required by each component path coverage criterion is often different.

A. Direct Component Path Coverage Criterion

Definition 3 For a component path π_{CP}: $C_s \rightarrow C_{s+1} \rightarrow \ldots \rightarrow C_t$ in CIG of the C2-style architecture, if each $(C_i, C_j) \in e_{Conn-Conn}$ for i = s+1, s+2, …, t-2, j = i+1, i+2, …, t-1, we call the π_{CP} to satisfy the direct component path coverage criterion, denoted as $DCPCC$.

Intuitively, the $DCPCC$ requires that all π_{CP} of length 2 will be covered. For example in Fig. 2, for π_{CP}: LayoutManager → GLConn → GraphicsBinding, its length is 2, according to $DCPCC$, this π_{CP} will be covered. Similarly, LayoutManager → LTConn → TAConn → StatusArtist also will be covered.

Let Π_{DCP} represents the set of component paths that covered by test suite TS on $DCPCC$, $||\Pi_{DCP}||$ represents the number of elements in Π_{DCP}, $||EP(\Pi_{DCP}(CIG))||$ represents the number of component paths in CIG on $DCPCC$, then component path coverage rate on $DCPCC$ is calculated as follows:

$$R_{DCP} = \frac{||\Pi_{DCP}||}{||EP(\Pi_{DCP}(CIG))||} \times 100\% \qquad (1)$$

B. Indirect Component Path Coverage Criterion

Definition 4 For a component path π_{CP}: $C_s \rightarrow C_{s+1} \rightarrow \ldots \rightarrow C_t$ in CIG of the C2-style architecture, if each $(C_i, C_j) \in e_{Conn-Comp} \vee e_{Comp-Conn} \vee e_{Conn-Conn}$ for i = s+1, s+2, …, t-2, j = i+1, i+2, …, t-1, we call the π_{CP} to satisfy the indirect component path coverage criterion, denoted as $ICPCC$.

Note that the $ICPCC$ requires that all length of π_{CP} is greater than or equal to 2 will be covered. For example in Fig. 2, for π_{CP}: LayoutManager → LLConn → TileArtist → TAConn → StatusArtist → ALAConn → LAConn → ClockLogic, its length is 7, according to $ICPCC$, this π_{CP} will be covered.

Let Π_{ICP} represents the set of component paths that covered by test suite TS on $ICPCC$, $||\Pi_{ICP}||$ represents the number of elements in Π_{ICP}, $||EP(\Pi_{ICP}(CIG))||$ represents the number of component paths on $ICPCC$ in CIG, then component path coverage rate on $ICPCC$ is calculated as follows:

$$R_{ICP} = \frac{||\Pi_{ICP}||}{||EP(\Pi_{ICP}(CIG))||} \times 100\% \qquad (2)$$

C. Length-N Component Path Coverage Criterion

Because CIG has many start nodes and terminal nodes, and a fixed length of π_{CP} is not necessarily between any two nodes. As the length of the π_{CP} increases, the number of possible contexts also increases.

Definition 5 For a component path π_{CP}: $C_s \rightarrow C_{s+1} \rightarrow \ldots \rightarrow C_t$ in CIG of the C2-style architecture, if the length of π_{CP} is less than or equal to N for i = s+1, s+2, …, t-2, j = i+1, i+2, …, t-1, N is a natural number greater than or equal to 2, we call the π_{CP} to satisfy the Length-N component path coverage criterion, denoted as L_NCPCC.

Note, the L_NCPCC requires that all length of π_{CP} greater than or equal to 2 will be covered. For example in Fig. 2, we can see that there are six π_{CP}s from component LayoutManager on L_3CPCC are shown as follows.

LayoutManager → LTConn → TAConn → StatusArtist
LayoutManager → LTConn → TAConn → ChuteArtist
LayoutManager → LTConn → TAConn → WellArtist
LayoutManager → LTConn → TAConn → PaletteArtist
LayoutManager → LTConn → TileArtist
LayoutManager → GLConn → GraphicsBinding

Let Π_{L_NCP} represents the set of component paths that covered by test suite TS on L_NCPCC, $||\Pi_{L_NCP}||$ represents the number of elements in Π_{L_NCP}, $||EP(\Pi_{L_NCP}(CIG))||$ represents the number of component paths on L_NCPCC in CIG, then component path coverage rate on L_NCPCC is calculated as follows:

$$R_{L_NCP} = \frac{||\Pi_{L_NCP}||}{||EP(\Pi_{L_NCP}(CIG))||} \times 100\% \qquad (3)$$

D. Component Path with Node-Sequence Coverage Criterion

Definition 6 For a component path π_{CP}: $C_s \rightarrow C_{s+1} \rightarrow \ldots \rightarrow C_t$, and for any node sequence $C_i, C_j, \ldots, C_k \in$ V for i, j, …, k = s+1, s+2, …, t-1 in CIG of the C2-style architecture, if the π_{CP} covers all nodes and nodes $C_i, C_j, \ldots, C_k$ reachable from C_s to C_t, we call the π_{CP} to satisfy the component path with node-sequence coverage criterion, denoted as $CPNSCC$.

Notice that every node of node sequence in π_{CP} doesn't all reachable, because some nodes in node sequence may not be on the component path from C_s to C_t. If the node sequence is on the component path from C_s to C_t, then the π_{CP} satisfies the $CPNSCC$.

For example in Fig. 2, we can see that there are two π_{CP}s from component GraphicsBinding to ChuteADT passing through two nodes LayoutManager and StatusArtist are shown as follows.

GraphicsBinding → GLConn → LayoutManager → LTConn → TileArtist → TAConn → StatusArtist → ALAConn → LAConn → ChuteADT

GraphicsBinding → GLConn → LayoutManager → LTConn → TAConn → StatusArtist → ALAConn → LAConn → ChuteADT

Let Π_{CPNS} represents the set of component paths that covered by test suite TS on $CPNSCC$, $||\Pi_{CPNS}||$ represents the number of elements in Π_{CPNS}, $||EP(\Pi_{CPNS}(CIG))||$ represents the number of component paths on $CPNSCC$ in CIG, then component path coverage rate on $CPNSCC$ is calculated as follows:

$$R_{CPNS} = \frac{||\Pi_{CPNS}||}{||EP(\Pi_{CPNS}(CIG))||} \times 100\% \qquad (4)$$

E. Component Path with Edge-Sequence Coverage Criterion

Definition 7 For a component path π_{CP}: $C_s \rightarrow C_{s+1} \rightarrow \ldots \rightarrow C_t$, and for any edge sequence $e_{C_{i_1},C_{j_1}}$, $e_{C_{i_2},C_{j_2}}$, …, $e_{C_{i_k},C_{j_k}} \in$ E for i_1, j_1, i_2, j_2, …, i_k, j_k = s+1, s+2, …, t-1 in CIG of the C2-style architecture, if the π_{CP} covers all nodes and edges $e_{C_{i_1},C_{j_1}}$, $e_{C_{i_2},C_{j_2}}$, …, $e_{C_{i_k},C_{j_k}}$ reachable from C_s to C_t, we call the π_{CP} to satisfy the component path with edge-sequence coverage criterion, denoted as $CPESCC$.

Notice that every edge of edge-sequence in π_{CP} doesn't all reachable, because some edges in edge-sequence may not be on the component path from C_s to C_t. If the edge sequence is on the component path from C_s to C_t, then the π_{CP} satisfies the $CPESCC$.

For example in Fig. 2, component path set on $CPNSCC$ doesn't cover edge (LLConn, LAConn), so, on the basis of component path on $CPNSCC$, adding a component path can satisfy $CPESCC$. Thus, we can see that there are four π_{CP}s from component LayoutManager to ChuteADT passing through two edges $e_{TileArtist,TAConn}$ and $e_{ALAConn,LAConn}$ are shown as follows.

LayoutManager → LTConn → TileArtist → TAConn → StatusArtist → ALAConn → LAConn → ChuteADT

LayoutManager → LTConn → TileArtist → TAConn → ChuteArtist → ALAConn → LAConn → ChuteADT

LayoutManager → LTConn → TileArtist → TAConn → WellArtist → ALAConn → LAConn → ChuteADT

LayoutManager → LTConn → TileArtist → TAConn → PaletteArtist → ALAConn → LAConn → ChuteADT

Let Π_{CPES} represents the set of component paths that covered by test suite TS on $CPESCC$, $||\Pi_{CPES}||$ represents the number of elements in Π_{CPES}, $||EP(\Pi_{CPES}(CIG))||$ represents the number of component paths on $CPESCC$ in CIG, then component path coverage rate on $CPESCC$ is calculated as follows:

$$R_{CPES} = \frac{||\Pi_{CPES}||}{||EP(\Pi_{CPES}(CIG))||} \times 100\% \qquad (5)$$

V. The Relationship Among Component Path Coverage Criteria

In this section, we prove the relationship among component path coverage criteria at first, and then give the subsumption relationships among these component path coverage criteria.

A. The Properties of Component Path Coverage Criteria

Theorem 1 The test suite that satisfying the $L_{N+1}CPCC$ also satisfies the L_NCPCC, and vice versa.

Proof: Let T_{L_N} be test suite that satisfies the L_NCPCC, $T_{L_{N+1}}$ be test suite that satisfies the $L_{N+1}CPCC$. Suppose, the test suite that satisfies $L_{N+1}CPCC$ doesn't satisfies L_NCPCC, that there exists a subpath π_{L_NCP} of length N, make that π_{L_NCP} is covered by T_{L_N} and doesn't cover by $T_{L_{N+1}}$. If π_{L_NCP} is a complete path, according to definition 2, it is a $\pi_{L_{N+1}CP}$. Because, $T_{L_{N+1}}$ satisfies the $\pi_{L_{N+1}CP}$ coverage, thus, there exists a test suite $ts_i \in TS_{L_{N+1}}$, make that the π_{L_NCP} is covered by t_i, that is conflict. If π_{L_NCP} doesn't a complete path, there exist predecessor nodes or successor nodes of π_{L_NCP}, let it is C_k. Thus, π_{L_NCP} + C_k is a $\pi_{L_{N+1}CP}$. Because $T_{L_{N+1}}$ satisfies the $L_{N+1}CPCC$, so, there exists a test suite $ts_i \in TS_{L_{N+1}}$, make that the π_{L_NCP} + C_k is covered by t_i. According to definition 2, π_{L_NCP} is also covered by t_i, that is conflict. Thus, the test suite that satisfying the $L_{N+1}CPCC$ also satisfies the L_NCPCC.

On the contrary, take Fig. 2 for example, where the corresponding test suite of π_{CP} LayoutManager → LTConn → TileArtist can covered by π_{L_2CP}, but for π_{L_3CP} LayoutManager → LTConn → TAConn → StatusArtist, this test suite doesn't cover π_{L_3CP}. Thus, the test suite that satisfying the L_NCPCC also satisfies the $L_{N+1}CPCC$.

Theorem 2 For given an C2-style architecture, the test suite that satisfying the $ICPCC$ also satisfies the $CPESCC$.

Proof: According to definition 4, the component path generated by $ICPCC$ will cover all edges $e_{C_{i_1},C_{j_1}}$, $e_{C_{i_2},C_{j_2}}$, …, $e_{C_{i_k},C_{j_k}}$ from C_s to C_t, but, the component path with edge-sequence set generated by $CPESCC$ will cover part edges of edges $e_{C_{i_1},C_{j_1}}$, $e_{C_{i_2},C_{j_2}}$, …, $e_{C_{i_k},C_{j_k}}$. Thus, the test suite that satisfying the $ICPCC$ also satisfies the $CPESCC$.

Theorem 3 For given an C2-style architecture, the test suite that satisfying the $CPESCC$ also satisfies the $CPNSCC$.

Proof: According to definition 7, the component path with edge-sequence set generated by $CPESCC$ will cover all edges $e_{C_{i_1},C_{j_1}}$, $e_{C_{i_2},C_{j_2}}$, …, $e_{C_{i_k},C_{j_k}}$ from C_s to C_t, but, the component path with node-sequence set generated by $CPNSCC$ will cover part nodes of nodes C_{i_1}, C_{j_1}, C_{i_2}, C_{j_2}, …, C_{i_k}, C_{j_k}. Thus, the test suite that satisfying the $ICPCC$ also satisfies the $CPESCC$.

Theorem 4 The test suite that satisfying the $CPESCC$ also satisfies the $DCPCC$.

Proof: According to definition 7, the component path generated by $CPESCC$ will coverage all edges $e_{Conn-Conn}$, $e_{Comp-Conn}$, and $e_{Conn-Comp}$ from C_s to C_t, but, the component path generated by $DCPCC$ is a subset of these component paths with edge-sequence. Thus, the test suite that satisfying the $CPESCC$ also satisfies the $DCPCC$.

Theorem 5 For given an C2-style architecture, the test suite that satisfying the $L_\infty CPCC$ is equivalent to the $ICPCC$.

Proof: When the length of π_{CP} is ∞, according to definition 2, all indirect component paths can be considered as a special L_NCPCC. At the moment, the test suite that satisfies L_NCPCC will cover all indirect component paths. So, make that the test suite that satisfying L_NCPCC also satisfies $ICPCC$. On the contrary, the test suite that covering all indirect component paths of software architecture also cover π_{CP_∞}. Thus, when the length of π_{CP} is ∞, the $L_\infty CPCC$ is equivalent to the $ICPCC$.

Theorem 6 For given an C2-style architecture, the test suite that satisfying the $L_K CPCC$ also satisfies the $CPESCC$.
Proof: When the length of π_{CP} is ∞, according to definition 2, all indirect component paths can be considered as a special $L_K CPCC$. At the moment, the test suite that satisfies $L_N CPCC$ will cover all indirect component paths. So, make that the test suite that satisfying $L_K CPCC$ also satisfies $CPESCC$. On the contrary, the test suite that covering all indirect component paths of software architecture also cover π_{CP_∞}. Thus, when the length of π_{CP} is ∞, the $L_\infty CPCC$ is equivalent to the $ICPCC$.

According to Theorem 6, we also can prove the properties of component path coverage criteria as follows:

- There exists a positive integer J, and J $<$ K, test suite that satisfying the $L_J CPCC$ also satisfies the $CPNSCC$.
- There exists a positive integer I, and I $<$ J, test suite that satisfying the $L_I CPCC$ also satisfies the $DCPCC$.

B. Subsumption Relationships Among Component Path Coverage Criteria

Different test coverage criteria have different degree of strict requirements on test quality, and have the ability to reveal faults and test overhead. Therefore, [24], [25], [26] et al. defined the subsumption relationships between test coverage criteria, then discussed the basis properties of subsumption relationships, and elaborated the relationship between testing adequacy and the ability to reveal faults by formal theory analysis and proof. We discuss the subsumption relationships among component path coverage criteria.

According to the definition [25], for two coverage criteria C_1 and C_2, the test suite t $\in$ T satisfies test coverage criterion C_1, represents as t $\downarrow C_1$. If $\forall$ t $\in$ T, there exists test suite t $\downarrow C_1 \Rightarrow$ t $\downarrow C_2$, then the coverage criterion C_1 subsumes the coverage criterion C_2.

We consider that this definition is also suitable for five component path coverage criteria. A component path coverage criterion $CPCC_a$ subsumes another component path coverage criterion $CPCC_b$, if and only if there exists test suite ts $\in$ TS, it makes ts $\downarrow CPCC_a \Rightarrow$ ts $\downarrow CPCC_b$. From the theoretical point of view, strength can be analyzed by the subsumption relationships. Based on subsumption relationships, we analyze the hierarchy relationships among the π_{CP} coverage criteria. According to Zhu [24], the subsumption relation is perhaps the property that we know best about adequacy criteria, although not all of them can be easily placed in the hierarchy, such as specification-based criteria. Zhu has also shown that under certain circumstance the subsumption relation can provide information to compare the effectiveness of the criteria.

The subsumption hierarchy relationships for the component path coverage criteria presented is shown in Fig. 3, where the bidirectional arrow represents the equivalence relationship, the solid arrow represents the subsumption relationship, the dotted arrow represents the subsumption relationship with several levels. Meanwhile, infinite, K, J, and I are positive integers, and satisfy infinite $<$ K $<$ J $<$ I.

From Fig. 3, we can see that, the test suite ts that covers all the indirect component paths certainly covers all the component paths with edge-sequence in C2-style architecture, so

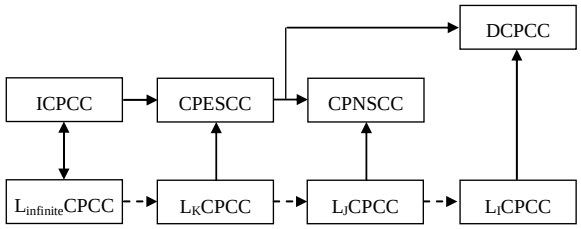

Fig. 3. Subsumption relationships among component path coverage criteria

the indirect component path coverage criterion subsumes the component path with edge-sequence coverage criteria, that is the $ICPCC$ subsumes the $CPESCC$. In the same way, the $CPESCC$ subsumes the $CPNSCC$ and $DCPCC$. Meanwhile, the $L_{infinite}CPCC$ is equivalent to the $ICPCC$ and the $L_{infinite}CPCC$ subsumes the $L_K CPCC$ with several level. The practical issue for testers, then, is to determine which component path coverage criterion to select for a particular situation. This includes determining how to balance such factors as effectiveness and cost.

VI. Algorithms for Component Path Coverage Rate

We now present algorithms to calculate the component path coverage rate using component path coverage criteria. Where algorithms to determine R_{DCP}, R_{ICP}, and $R_{L_N CP}$ have been discussed in [6]. In this section, we propose two algorithms to calculate R_{CPNS} and R_{CPES} on $CPNSCC$ and $CPESCC$ from beginning node C_s to stopping node C_t. The two algorithms contain four procedures as follows:

- Procedure isConnected(C_s, C_i, C_j, …, C_k, C_t): is used to determine the connectivity of nodes C_i, C_j, …, C_k. In these nodes, select the closest node from C_s as a beginning node, depth first traversal of the CIG, if the other nodes can be traversed, it is connected; conversely, it is not connected.
- Procedure Sequence(C_1, C_2, …, C_n): is used to obtain the sequence of intermediate nodes C_1, C_2, …, C_n of nodes C_i, C_j, …, C_k in order to generate an intermediate node sequence. The beginning node of node sequence is C_1, and the stopping node of node sequence is C_n.
- Procedure Prefix(C_{k1}, π_{Pk1}): is used to obtain the prefix of node C_{k1} of component path π_{Pk1}.
- Procedure Postfix(C_{k2}, π_{Pk1}): is used to obtain the postfix of node C_{k2} of component path π_{Pk1}.

A. Algorithm for R_{CPNS}

Algorithm RCPNSA can be used to calculate the component path with node-sequence coverage rate. The main idea of RCPNSA algorithm can be briefly stated as follows: Firstly, it calls isConnected() to determine the connectivity among beginning node C_s, intermediate nodes C_i, C_j, …, C_k, and stopping node C_t. Then, it calls Sequence() to determine the direction of C_i, C_j, …, and C_k from C_s to C_t and represents as C_1, C_2, …, C_n. Finally, it calls procedure CPNSA() to generate component path coverage set from C_s to C_t passing through nodes C_1, C_2, …, and C_n.

The method of calculating R_{CPNS} is shown as Algorithm 1.

Algorithm 1 RCPNSA(CIG, C_s, C_i, C_j, …, C_k, C_t, R_{CPNS})

Require: CIG, C_s is beginning component, C_i, C_j, …, C_k are component or connector nodes, C_t is stopping component.

Ensure: R_{CPNS} is component path coverage rate on $CPNSCC$.

Begin
1 **if** (!isConnected(C_s, C_i, C_j, …, C_k, C_t)) **then**
2 **return**;
3 **end if**
4 Sequence(C_1, C_2, …, C_n) ← C_i, C_j, …, C_k;
5 CPNS' = ∅;
6 CPNS' = CPNSA(C_s, C_1, C_2, …, C_n, C_t)
7 Output R_{CPNS} = ||CPNS'|| / ||CPNS|| × 100%;
8 **return** R_{CPNS};
Procedure CPNSA(C_s, C_1, C_2, …, C_n, C_t)
9 CPNSSet = ∅;
10 TempP1 = MidP(C_s, C_1);
11 CPNSSet = CPNSSet + TempP1;
12 **for** (k = 1; k <= n; k ++)
13 TempP2 = MidP(C_k, C_{k+1});
14 CPNSSet = CPNSSet + TempP2;
15 **end for**
16 TempP3 = MidP(C_n, C_t);
17 CPNSSet = CPNSSet + TempP3;
18 **return** CPNSSet;
Procedure MidP(C_i, C_j)
19 π_P = ∅;
20 **for** (k1 = i; k1 < j; k1 ++)
21 add C_{k1} to π_P;
22 **for** (k2 = k1 + 1; k2 <= j; k2 ++)
23 **if** ($e_{C_{k1},C_{k2}}$ ∈ E ∧ C_{k2} ∉ π_P) **then**
24 add C_{k2} to π_P;
25 **end if**
26 **end for**
27 **end for**
28 **return** π_P;
End RCPNSA

We employ the CIG shown in Fig. 2 to demonstrate algorithm RCPNSA. Let us consider examples showing the computation of component path coverage rate from component GraphicsBinding to component ClockLogic passing through two nodes ALAConn and TileArtist on $CPNSCC$. That is C_s = GraphicsBinding, C_t = ClockLogic.

Firstly, according to step 1, isConnected(GraphicsBinding, ALAConn, TileArtist, ClockLogic) = true, so, there exists π_{CP} from GraphicsBinding to ClockLogic passing through ALAConn and TileArtist. According to step 4, Sequence(GraphicsBinding, TileArtist, ALAConn, ClockLogic) ← GraphicsBinding, ALAConn, TileArtist, ClockLogic.

Secondly, calls procedure CPNSA(GraphicsBinding, TileArtist, ALAConn, ClockLogic) to generate component path coverage set. According to step 10, computing TempP1 = MidP(GraphicsBinding, TileArtist) by calling MidP(). According to steps 19-27 of MidP(), we get π_P = {GraphicsBinding → GLConn → LayoutManager → LTConn → TileArtist}. After return, CPNSSet = {GraphicsBinding → GLConn → LayoutManager → LTConn → TileArtist}.

Thirdly, according to steps 12-15, computing TempP2 from TileArtist to ALAConn by calling MidP(). We get CPNSet = {GraphicsBinding → GLConn → LayoutManager → LTConn → TileArtist → TAConn → StatusArtist → ALAConn, GraphicsBinding → GLConn → LayoutManager → LTConn → TileArtist → TAConn → ChuteArtist → ALAConn, GraphicsBinding → GLConn → LayoutManager → LTConn → TileArtist → TAConn → WellArtist → ALAConn, GraphicsBinding → GLConn → LayoutManager → LTConn → TileArtist → TAConn → PaletteArtist → ALAConn}.

Finally, computing TempP3 from ALAConn to ClockLogic by calling MidP(). According to steps 19-27, we get TempP3 = {ALAConn → LAConn → ClockLogic}.

Thus, according to step 6, CNPS' = {GraphicsBinding → GLConn → LayoutManager → LTConn → TileArtist → TAConn → StatusArtist → ALAConn → LAConn → ClockLogic, GraphicsBinding → GLConn → LayoutManager → LTConn → TileArtist → TAConn → ChuteArtist → ALAConn → LAConn → ClockLogic, GraphicsBinding → GLConn → LayoutManager → LTConn → TileArtist → TAConn → WellArtist → ALAConn → LAConn → ClockLogic, GraphicsBinding → GLConn → LayoutManager → LTConn → TileArtist TileArtist → TAConn → PaletteArtist → ALAConn → LAConn → ClockLogic}.

Therefore, the number of component paths on $CPNSCC$ from GraphicsBinding to ClockLogic passing through LAConn and TileArtist is 4.

While the all number of component paths on $CPNSCC$ for GraphicsBinding to ClockLogic passing through two nodes is 196. Hence, according to step 7, the R_{CPNS} = 4 / 196 × 100% = 2.04%.

B. Algorithm for R_{CPES}

Algorithm RCPESA can be used to calculate the component path with edge-sequence coverage rate. The main idea of RCPESA algorithm can be briefly stated as follows: Firstly, it calls isConnected() to determine the connectivity among beginning node C_s, intermediate nodes C_{i_1}, C_{j_1}, C_{i_2}, C_{j_2}, …, C_{i_k}, C_{j_k}, and stopping node C_t. It continues to call Sequence() to determine the direction of C_{i_1}, C_{j_1}, C_{i_2}, C_{j_2}, …, C_{i_k}, and C_{j_k} from C_s to C_t and represents as C_1, C_2, …, C_n. Then, it calls procedure CPNSA() to generate component path coverage set from C_s to C_t passing through nodes C_1, C_2, …, and C_n. Finally, it calls procedure CPESA() to generate component path coverage set that has not been covered edges by procedure CPNSA().

The method of calculating R_{CPES} is shown as Algorithm 2.

We employ the CIG shown in Fig. 2 to demonstrate algorithm RCPESA. Let us consider examples showing the computation of component path coverage rate from component WellADT to component GraphicsBinding passing through two edges $e_{ALAConn,LAConn}$ and $e_{LayoutManager,LTConn}$ on $CPESCC$. That is C_s = WellADT, C_t = GraphicsBinding.

Firstly, according to step 1, isConnected(WellADT, ALAConn, LAConn, LayoutManager, LTConn, GraphicsBinding) = true, so, there exists π_{CP} from WellADT to GraphicsBinding passing through $e_{ALAConn,LAConn}$

Algorithm 2 RCPESA(CIG, C_s, $e_{C_{i_1},C_{j_1}}$, $e_{C_{i_2},C_{j_2}}$, …, $e_{C_{i_k},C_{j_k}}$, C_t, R_{CPES})

Require: CIG, C_s is beginning component, $e_{C_{i_1},C_{j_1}}$, $e_{C_{i_2},C_{j_2}}$, …, $e_{C_{i_k},C_{j_k}}$ are edges, C_t is stopping component.
Ensure: R_{CPES} is component path coverage rate on $CPESCC$.
Begin
1 **if** (!isConnected(C_s, C_{i_1}, C_{j_1}, C_{i_2}, C_{j_2}, …, C_{i_k}, C_{j_k}, C_t)) **then**
2 **return**;
3 **end if**
4 Sequence(C_1, C_2, …, C_n) ← C_{i_1}, C_{j_1}, C_{i_2}, C_{j_2}, …, C_{i_k}, C_{j_k};
5 CPES' = ∅;
6 CPNS' = ∅;
7 CPNS' = CPNSA(C_s, C_1, C_2, …, C_n, C_t);
8 CPES' = CPESA(C_s, C_1, C_2, …, C_n, C_t, CPNS');
9 Output R_{CPES} = ||CPES'|| / ||CPES|| × 100%;
10 **return** R_{CPES};
Procedure CPESA(C_s, C_i, C_j, …, C_k, C_t, CPNS')
11 CPESSet = ∅;
12 **for** (k1 = 1; k1 <= |CPNS'|; k1 ++)
13 **if** ($\pi_{Pk1} \in$ CPNS') **then**
14 CPESSet = CPESSet + π_{Pk1};
15 **for** (k2 = 1; k2 <= $|E|$; k2 ++)
16 **if** ($e_{C_{k1},C_{k2}} \in$ E ∧ $C_{k1} \in \pi_{Pk1}$ ∧ $C_{k2} \in \pi_{Pk1}$ ∧ $e_{C_{k1},C_{k2}} \notin$ CPESSet) **then**
17 TempP1 = Prefix(C_{k1}, π_{Pk1});
18 TpmpP2 = Postfix(C_{k2}, π_{Pk1});
19 CPESSet = TempP1 + C_{k1} + C_{k2} + TempP2;
20 **end if**
21 **end for**
22 **end if**
23 **end for**
24 **return** CPESSet;
End RCPESA

and $e_{LayoutManager,LTConn}$. According to step 4, Sequence(WellADT, LAConn, ALAConn, LTConn, LayoutManager, GraphicsBinding) ← WellADT, ALAConn, LAConn, LayoutManager, LTConn, GraphicsBinding.

Secondly, calls procedure CPNSA(WellADT, LAConn, ALAConn, LTConn, LayoutManager, GraphicsBinding) to generate component path coverage set CPNS' = {WellADT → LAConn → ALAConn → StatusArtist → TAConn → TileArtist → LTConn → LayoutManager → GLConn → GraphicsBinding, WellADT → LAConn → ALAConn → ChuteArtist → TAConn → TileArtist → LTConn → LayoutManager → GLConn → GraphicsBinding, WellADT → LAConn → ALAConn → WellArtist → TAConn → TileArtist → LTConn → LayoutManager → GLConn → GraphicsBinding, WellADT → LAConn → ALAConn → Palette → TAConn → TileArtist → LTConn → LayoutManager → GLConn → GraphicsBinding} on $CPNSCC$. According to step 8, calls procedure CPESA(WellADT, LAConn, ALAConn, LTConn, LayoutManager, GraphicsBinding, CPNS') to generate component path coverage set from WellADT to GraphicsBinding on $CPESCC$.

Thirdly, according to steps 13-14, π_{P1} = WellADT → LAConn → ALAConn → StatusArtist → TAConn → TileArtist → LTConn → LayoutManager → GLConn → GraphicsBinding and CPESSet = {WellADT → LAConn → ALAConn → StatusArtist → TAConn → TileArtist → LTConn → LayoutManager → GLConn → GraphicsBinding}. According to step 16, $e_{TAConn,LTConn} \in$ E ∧ TAConn $\in \pi_{P1}$ ∧ LTConn $\in \pi_{P1}$ ∧ $e_{TAConn,LTConn} \notin$ CPESSet, so, according to steps 17-18, obtains the prefix TempP1 = WellADT → LAConn → ALAConn → ChuteArtist of TAConn of π_{P1} and the postfix TempP2 = LayoutManager → GLConn → GraphicsBinding of LTConn of π_{P1}. Then, connects the prefix with TAConn, LTConn, and the postfix to generate path WellADT → LAConn → ALAConn → StatusArtist → TAConn → LTConn → LayoutManager → GLConn → GraphicsBinding and adds to CPESSet. According to step 19, CPESSet = {WellADT → LAConn → ALAConn → StatusArtist → TAConn → TileArtist → LTConn → LayoutManager → GLConn → GraphicsBinding, WellADT → LAConn → ALAConn → StarusArtist → TAConn → LTConn → LayoutManager → GLConn → GraphicsBinding}.

Fourth, according to steps 13-14, we get π_{P2} = WellADT → LAConn → ALAConn → ChuteArtist → TAConn → TileArtist → LTConn → LayoutManager → GLConn → GraphicsBinding and CPESSet = {WellADT → LAConn → ALAConn → StatusArtist → TAConn → TileArtist → LTConn → LayoutManager → GLConn → GraphicsBinding, WellADT → LAConn → ALAConn → StarusArtist → TAConn → LTConn → LayoutManager → GLConn → GraphicsBinding, WellADT → LAConn → ALAConn → ChuteArtist → TAConn → TileArtist → LTConn → LayoutManager → GLConn → GraphicsBinding}. According to step 16, because $e_{TAConn,LTConn} \in$ E ∧ TAConn $\in \pi_{P2}$ ∧ $e_{TAConn,LTConn} \in$ CPESSet, so, $e_{TAConn,LTConn}$ doesn't require to be added to CPESSet.

Similarly, the other component paths generation are the same.

Thus, the component path coverage set on CPESCC is CPES' = {WellADT → LAConn → ALAConn → StatusArtist → TAConn → TileArtist → LTConn → LayoutManager → GLConn → GraphicsBinding, WellADT → LAConn → ALAConn → StarusArtist → TAConn → LTConn → LayoutManager → GLConn → GraphicsBinding, WellADT → LAConn → ALAConn → ChuteArtist → TAConn → TileArtist → LTConn → LayoutManager → GLConn → GraphicsBinding, WellADT → LAConn → ALAConn → WellArtist → TAConn → TileArtist → LTConn → LayoutManager → GLConn → GraphicsBinding, WellADT → LAConn → ALAConn → PaletteArtist → TAConn → TileArtist → LTConn → LayoutManager → GLConn → GraphicsBinding}.

Therefore, the number of component paths from WellADT to GraphicsBinding passing through $e_{ALAConn,LAConn}$ and $e_{LayoutManager,LTConn}$ is 4.

While the all number of component paths from WellADT to GraphicsBinding passing through two edges is 64. Hence, according to step 9, the R_{CPES} = 4 / 64 × 100% = 6.25%.

VII. Experimental Studies

In order to verify the effectiveness and performance of the proposed component path coverage criteria, we carry out lots of experiments. In this section, we present an analysis of

TABLE I
TOTAL NUMBER OF COMPONENT PATHS FOR DCPCC, ICPCC, AND L_NCPCC

Component name	DCPCC	ICPCC	L_NCPCC							
			N=2	N=3	N=4	N=5	N=6	N=7	N=8	N=9
GraphicsBinding	1	50	1	1	2	6	10	10	30	50
LayoutManager	6	50	2	6	10	10	30	50	50	50
TileArtist	5	26	5	5	6	26	26	26	26	26
StatusArtist	7	10	1	7	8	9	10	10	10	10
ChuteArtist	7	10	1	7	8	9	10	10	10	10
WellArtist	7	10	1	7	8	9	10	10	10	10
PaletteArtist	7	10	1	7	8	9	10	10	10	10
StatusLogic	7	17	2	7	17	17	17	17	17	17
NextTilePlacingLogic	5	5	5	5	5	5	5	5	5	5
TileMatchLogic	6	6	6	6	6	6	6	6	6	6
RelativePosLogic	6	6	6	6	6	6	6	6	6	6
ClockLogic	8	30	3	8	10	14	18	22	26	30
StatusADT	8	30	3	8	10	14	18	22	26	30
ChuteADT	8	30	3	8	10	14	18	22	26	30
WellADT	8	30	3	8	10	14	18	22	26	30
PaletteADT	8	30	3	8	10	14	18	22	26	30

TABLE II
TOTAL NUMBER OF COMPONENT PATHS FOR CPNSCC AND CPESCC

Component name	CPNSCC								CPESCC								
	N=1	N=2	N=3	N=4	N=5	N=6	N=7	N=8	N=1	N=2	N=3	N=4	N=5	N=6	N=7	N=8	N=9
GraphicsBinding	184	603	1161	1420	1124	560	160	20	390	1387	2924	4001	3668	2244	880	200	20
LayoutManager	134	312	404	300	120	20	-	-	292	758	1120	1004	540	160	20	-	-
TileArtist	88	123	81	20	-	-	-	-	114	211	204	101	20	-	-	-	-
StatusArtist	19	18	11	5	1	-	-	-	35	50	40	20	7	1	-	-	-
ChuteArtist	19	18	11	5	1	-	-	-	35	50	40	20	7	1	-	-	-
WellArtist	19	18	11	5	1	-	-	-	35	50	40	20	7	1	-	-	-
PaletteArtist	19	18	11	5	1	-	-	-	35	50	40	20	7	1	-	-	-
StatusLogic	32	30	11	-	-	-	-	-	39	47	45	10	-	-	-	-	-
NextTilePlacingLogic	5	-	-	-	-	-	-	-	10	1	-	-	-	-	-	-	-
TileMatchLogic	6	-	-	-	-	-	-	-	12	6	-	-	-	-	-	-	-
RelativePosLogic	6	-	-	-	-	-	-	-	12	6	-	-	-	-	-	-	-
ClockLogic	89	206	322	344	248	116	32	4	165	464	833	1006	837	480	152	40	4
StatusADT	89	206	322	344	248	116	32	4	165	464	833	1006	837	480	152	40	4
ChuteADT	89	206	322	344	248	116	32	4	165	464	833	1006	837	480	152	40	4
WellADT	89	206	322	344	248	116	32	4	165	464	833	1006	837	480	152	40	4
PaletteADT	89	206	322	344	248	116	32	4	165	464	833	1006	837	480	152	40	4

the experimental results proposed component path coverage criteria in the previous section.

A. Experimental Results

We apply our method to the KLAX system as a case study for testing purposes, and statistics the total number of component paths. The total number of component paths is shown in Table I-II. In Table I, the first column represents the component name of KLAX system, the second column represents the number of component paths for the first column component on $DCPCC$, the third column represents the number of component paths for the first column component on $ICPCC$, the fourth column represents the number of component paths of length N for the first column component on L_NCPCC. In Table II, the first column represents the component name of KLAX system, the second column represents the number of component paths with node N for the first column component on $CPNSCC$, the third column represents the number of component paths with edge N for the first column component on $CPESCC$, the symbol "-" means that there doesn't exist component path for component. For example, the number of component paths for component LayoutManager on $DCPCC$ is 6. The number of component paths for component WellArtist on $ICPCC$ is 10. The number of component paths of length 6 for component StatusArtist on L_NCPCC is 10. The number of component paths with node 1 for component GraphicsBinding on $CPNSCC$ is 184. The number of component paths with edge 2 for component StatusADT on $CPESCC$ is 464. There doesn't exist the component paths which the number of component paths with node greater than 8 and the number of component paths with edge greater than 9.

From the Table I-II, we can see that the total number of component paths on $ICPCC$ is greater than component paths for component on $DCPCC$. Note that $ICPCC$ subsumes on $DCPCC$, if all component paths on $ICPCC$ are tested, then so are all component paths on $DCPCC$.

TABLE III
COMPONENT PATH COVERAGE RATE FOR R_{DCP}, R_{ICP}, AND R_{L_NCP}

Component name	R_{DCP} (%)	R_{ICP} (%)	R_{L_NCP} (%)							
			N=2	N=3	N=4	N=5	N=6	N=7	N=8	N=9
GraphicsBinding	0.96	14.29	2.17	0.96	1.49	3.30	4.35	3.70	9.68	14.29
LayoutManager	5.77	14.29	4.35	5.77	7.46	5.49	13.04	18.52	16.13	14.29
TileArtist	4.81	7.43	10.87	4.81	4.48	14.29	11.30	9.63	8.39	7.43
StatusArtist	6.73	2.86	2.17	6.73	5.97	4.95	4.35	3.70	3.23	2.86
ChuteArtist	6.73	2.86	2.17	6.73	5.97	4.95	4.35	3.70	3.23	2.86
WellArtist	6.73	2.86	2.17	6.73	5.97	4.95	4.35	3.70	3.23	2.86
PaletteArtist	6.73	2.86	2.17	6.73	5.97	4.95	4.35	3.70	3.23	2.86
StatusLogic	6.73	4.86	4.35	6.73	12.69	9.34	7.39	6.30	5.48	4.86
NextTilePlacingLogic	4.81	1.43	10.87	4.81	3.73	2.75	2.17	1.85	1.61	1.43
TileMatchLogic	5.77	1.71	13.04	5.77	4.48	3.30	2.61	2.22	1.94	1.71
RelativePosLogic	5.77	1.71	13.04	5.77	4.48	3.30	2.61	2.22	1.94	1.71
ClockLogic	7.69	8.57	6.52	7.69	7.46	7.69	7.83	8.15	8.39	8.57
StatusADT	7.68	8.57	6.52	7.69	7.46	7.69	7.83	8.15	8.39	8.57
ChuteADT	7.69	8.57	6.52	7.69	7.46	7.69	7.83	8.15	8.39	8.57
WellADT	7.69	8.57	6.52	7.69	7.46	7.69	7.83	8.15	8.39	8.57
PaletteADT	7.69	8.57	6.52	7.69	7.46	7.69	7.83	8.15	8.39	8.57

The total number of component paths on $ICPCC$ is greater than the total number of component paths with node N on $CPNSCC$, and the total number of component paths with node N on $CPNSCC$ is greater than the total number of component paths on $DCPCC$. Note that component path on $ICPCC$ subsumes component path with node N on $CPNSCC$. The total number of component paths with edge N on $CPESCC$ is greater than the total number of component paths on $DCPCC$. Although most of the total number of indirect component paths are less than the total number of component paths with edge-sequence of edge N, the reason is that the component path with edge-sequence of edge N covers edges in indirect component path. So, the total number of component paths with edge-sequence of edge N is larger. The $CPESCC$ fills the gaps between $ICPCC$ and $DCPCC$. The total number of component paths with edge-sequence of edge N is greater than component paths with node-sequence of node N for component. Note that component path with edge-sequence of edge N subsumes component path with node-sequence of node N. The total number of component paths with edge-sequence of edge N decreases with increasing edge, a few component paths with edge-sequence of edge N grows with increasing node. The number of edges that the component path with edge-sequence needs to test should be more. The total number of component paths with edge-sequence of edge N increases with increasing node, and the total number of component paths with edge-sequence of edge N increases with decreasing node.

B. Experimental Results Analysis

Table III-IV report the component path coverage rate on $DCPCC$, $ICPCC$, L_NCPCC, $CPNSCC$, and $CPESCC$ for each component of KLAX system. In Table III, the first column represents the component name of KLAX system, the second column represents the component path coverage rate on $DCPCC$ for the first column component, the third column represents the component path coverage rate on $ICPCC$ for the first column component, the fourth column represents the component path of length N coverage rate on L_NCPCC for the first column component. In Table IV, the first column represents the component name of KLAX system, the second column represents the component path coverage rate on $CPNSCC$ for the first column component, the third column represents the component path coverage rate on $CPESCC$ for the first column component, the symbol "-" means that there doesn't exist component path coverage rate for component.

From the Table III-IV, we can see that for component GraphicsBinding, ClockLogic, StatusADT, ChuteADT, WellADT, and PaletteADT, the component path from length 9 to length 2 coverage rates on L_NCPCC decreases from 57.14% to 34.78%. However, for component StatusArtist, ChuteArtist, WellArtist, and PaletteArtist, the component path of length 2, 3, 4, 5, 6, 7, 8, and 9 coverage rates on L_NCPCC decreases from 26.92% to 8.70%. The reason is, component at the top/bottom of software architecture due to large number of components interacts with other levels, the number of component paths of length also increases relative, it's the component path coverage rate is relatively high. But the component at the middle level, its number of component paths is less than top/bottom component, that is the number of component paths will be decreased in middle level, making the component path coverage rate is relatively low. For component GraphicsBinding, ClockLogic, StatusADT, ChuteADT, WellADT, and PaletteADT, the component path coverage rate will increase as the number of passing through nodes increases. The component path with node-sequence of node 1, 2, 3, 4, 5, 6, 7, and 8 coverage rates on $CPNSCC$ increases from 64.45% to 100%. However, for component StatusArtist, ChuteArtist, WellArtist, and PaletteArtist, the component path coverage rate will decrease as the number of passing through nodes decreases. The component path with node 1, 2, 3, 4, and 5 coverage rate on $CPNSCC$ decreases from 7.80% to 0.16%. The same reason is, component at the top/bottom of software architecture due to large number

TABLE IV
COMPONENT PATH COVERAGE RATE FOR R_{CPNS} AND R_{CPES}

Component name	R_{CPNS} (%)								R_{CPES} (%)								
	N=1	N=2	N=3	N=4	N=5	N=6	N=7	N=8	N=1	N=2	N=3	N=4	N=5	N=6	N=7	N=8	N=9
GraphicsBinding	18.85	27.79	35.08	40.80	45.18	48.28	50.00	50.00	21.26	28.10	33.93	39.13	43.45	46.67	48.89	50.00	50.00
LayoutManager	13.73	14.38	12.21	8.62	4.82	1.72	-	-	15.92	15.36	13.00	9.82	6.40	3.33	0.11	-	-
TileArtist	9.02	5.67	2.45	0.57	-	-	-	-	6.22	4.27	2.37	0.99	0.24	-	-	-	-
StatusArtist	1.95	0.83	0.33	0.14	0.04	-	-	-	1.91	1.01	0.46	0.20	0.08	0.02	-	-	-
ChuteArtist	1.95	0.83	0.33	0.14	0.04	-	-	-	1.91	1.01	0.46	0.20	0.08	0.02	-	-	-
WellArtist	1.95	0.83	0.33	0.14	0.04	-	-	-	1.91	1.01	0.46	0.20	0.08	0.02	-	-	-
PaletteArtist	1.95	0.83	0.33	0.14	0.04	-	-	-	1.91	1.01	0.46	0.20	0.08	0.02	-	-	-
StatusLogic	3.28	1.38	0.30	-	-	-	-	-	2.13	0.95	0.52	0.10	-	-	-	-	-
NextTilePlacingLogic	0.51	-	-	-	-	-	-	-	0.55	0.02	-	-	-	-	-	-	-
TileMatchLogic	0.61	-	-	-	-	-	-	-	0.65	0.12	-	-	-	-	-	-	-
RelativePosLogic	0.61	-	-	-	-	-	-	-	0.65	0.12	-	-	-	-	-	-	-
ClockLogic	9.12	9.49	9.73	9.89	9.97	10.00	10.00	10.00	9.00	9.40	9.67	9.84	9.92	9.98	10.00	10.00	10.00
StatusADT	9.12	9.49	9.73	9.89	9.97	10.00	10.00	10.00	9.00	9.40	9.67	9.84	9.92	9.98	10.00	10.00	10.00
ChuteADT	9.12	9.49	9.73	9.89	9.97	10.00	10.00	10.00	9.00	9.40	9.67	9.84	9.92	9.98	10.00	10.00	10.00
WellADT	9.12	9.49	9.73	9.89	9.97	10.00	10.00	10.00	9.00	9.40	9.67	9.84	9.92	9.98	10.00	10.00	10.00
PaletteADT	9.12	9.49	9.73	9.89	9.97	10.00	10.00	10.00	9.00	9.40	9.67	9.84	9.92	9.98	10.00	10.00	10.00

of components interacts with other levels, the number of component paths of length also increases relative, it's the component path coverage rate is relatively high. However, the component at the middle level, its number of component paths is less than top/bottom component, that is the number of component paths will be a decrease in the middle level, making the component path coverage rate is relatively low. For GraphicsBinding, ClockLogic, StatusADT, ChuteADT, WellADT, and PaletteADT, the component path coverage rate will increase with the number of passing through edges increases. The component path with edge-sequence of node 1, 2, 3, 4, 5, 6, 7, 8, and 9 coverage rate on $CPESCC$ increases from 66.26% to 100%. However, for component StatusArtist, ChuteArtist, WellArtist, and PaletteArtist, the component path coverage rate will decrease as the number of passing through edges decreases. The component path with edge-sequence of edge 1, 2, 3, 4, 5, and 6 coverage rates on $CPESCC$ decreases from 7.64% to 0.08%. The reasons are the same ones.

VIII. CONCLUSION

A high quality software architecture needs to be tested as far as possible to ensure good interaction among components to avoid interaction faults. In order to solve this problem, on the basis of component path analysis, this paper proposes two component path coverage criteria for covering intermediate nodes and edges, that is component path with node-sequence coverage criterion and component path with edge-sequence coverage criterion. These component path coverage criteria define the adequacy of component path testing at several different levels. Meanwhile, we also discuss the subsumption relationships among component path coverage criteria. Two algorithms are given to compute component path coverage rate that can satisfy two component path coverage criteria. The experimental results show that testing effects are greatly determined by the selection of component path coverage criteria. Through the discussion of this paper, enriching and perfecting the existing software architecture coverage criteria. Software testers can get some conclusions that assist them to apply these coverage criteria, the difference among using different component path coverage criteria provide reference in practice. At the same time, so that we will do further research on the more powerful coverage criteria.

ACKNOWLEDGMENT

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.

REFERENCES

[1] H. Mei and J. R. Shen, "Progress of research on software architecture," *Journal of Software*, vol. 17, no. 6. pp. 1257-1275, 2006.
[2] H. Zhu, P. A. V. Hall and J. H. R. May, "Software unit test coverage and adequacy," *ACM Computing Surveys*, vol. 29, no. 4, pp. 366-427, 1997.
[3] H. Muccini, A. Bertolino and P. Inverardi, "Using software architecture for code testing," *IEEE Trans. Softw. Engi.*, vol. 30, no. 3, pp. 160-171, 2004.
[4] A. Bertolino, P. Inverardi and H. Muccini, "An explorative journey from architectural tests definition downto code test execution," in *Proceedings of the International Conference on Software Engineering*, May 2001, pp. 211-220.
[5] J. F. Chen, Y. S. Lu and H. H. Wang, "Component security testing approach based on extended chemical abstract machine," *International Journal of Software Engineering and Knowledge Engineering*, vol.22, no. 1, pp.59-83, 2012.
[6] L. J. Lun and X. Chi, "Component Dependency Path Coverage Criteria for C2-Style Architecture Testing," *IAENG International Journal of Computer Science*, vol. 42, no. 4, pp. 368-377, 2015.
[7] L. G. Yu and S. Ramaswamy, "Component dependency in object-oriented software,", *Journal of Computer Science and Technology*, vol. 22, no. 3, pp. 379-386, 2007.
[8] J. A. Stafford, A. L. Wolf and M. Caporuscio, "The application of dependence analysis to software architecture descriptions," *Formal Methods for Software Architectures, LNCS 2804*, 2003, pp. 52-62.
[9] M. R. Paige, "Program graphs, an algebra, and their implication for programming," *IEEE Trans. Softw. Engi.*, vol. 1, no. 3, pp. 286-291, 1975.
[10] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Engi.*, vol. 2, no. 4, pp. 308-320, 1976.
[11] E. F. Miller, "Tutorial: program testing techniques," in *Proceedings of IEEE Annual Computer Software and Applications Conference*, November 1977, pp. 107-120.

[12] J. S. Gourlay, "A mathematical framework for the investigation of testing," *IEEE Trans. Softw. Engi.*, vol. 9, no. 6, pp. 686-709, 1983.
[13] B. L. Li, Z. S. Li and J. C. Ni, "Research for test case generation based on Length_N criterion," *Journal of Sichuan University: Engineering Science Edition*, vol. 40, no. 3, pp. 132-137, 2008.
[14] Y. Li, Z. D. Su, L. Z. Wang and X. D. Li, "Steering symbolic execution to less traveled paths," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, October 2013, pp. 13-32.
[15] D. S. Rosenblum, "Adequate testing of component-based software," Technical Report, TR97-34, 1997.
[16] J. A. Stafford, D. J. Richardson and A. L. Wolf, "Chaining: a software architecture dependence analysis technique," Technical Report CU-CS-845-97, 1997.
[17] D. J. Richardson, J. A. Stafford and A. L. Wolf, "A formal approach to architecture-based software testing," Technical Report, 1998.
[18] Z. L. Jin and J. Offutt, "Deriving tests from software architectures," in *Proceedings of International Symposium on Software Reliability Engineering*, November 2001, pp. 308-313.
[19] N. L. Hashim, S. Ramakrishnan and H. W. Schmidt, "Architectural test coverage for component-based integration testing," in *Proceedings of International Conference on Quality Software*, October 2007, pp. 262-267.
[20] N. T. Richard, N. Medvidovic, K. M. Anderson, E. J. Whitehead and J. E. Robbins, "A component- and message-based architecture style for GUI software," *IEEE Trans. Softw. Engi.*, vol. 22, no. 6, pp. 390-406, 1996.
[21] M. Muccini, M. Dias and D. J. Richardson, "Systematic testing of software architectures in the C2 style," in *Fundamental Approaches to Software Engineering, LNCS 2984*, 2004, pp. 295-309.
[22] L. J. Lun, S. T. Wang, X. Chi and H. Xu, "Automatic generation of basis component path coverage for software architecture testing," *Computing and Informatics*, vol. 36, no. 2, pp. 386-404, 2017.
[23] J. B. Goodenough and S. L. Gerhart, "Toward a theory of test data selection," *IEEE Trans. Softw. Engi.*, vol. 1, no. 2, pp. 156-173, 1975.
[24] H. Zhu, "A formal analysis of the subsume relation between software test adequacy criteria," *IEEE Trans. Softw. Engi.*, vol. 22. no. 4, pp. 248-255, 1996.
[25] A. P. Mathur and W. E. Wong, "A formal evaluation of mutation and data flow based test adequacy criteria," http://citeseer.nj.nec.com/Mathor94formal.html.
[26] P. G. Frankl and E. Weyuker, "A formal analysis of the fault-detecting ability of testing methods," *IEEE Trans. Softw. Engi.*, vol. 19, no. 3, pp. 202-213, 1993.

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.

He is currently a professor in computer science and information engineering at Harbin Normal University of Harbin. He has published more than 70 papers in international and Chinese scientific journals. His research interests include software modeling, software analysis, empirical software engineering, software architecture, software testing, and software metrics.

Xin Chi was born in Harbin, Heilongjiang Province, China, in 1990. She received her B.S. degree in Computer Science and Technology from Harbin Normal University of Computer Science and Information Engineering, China, in 2013.

She has published more than 20 papers in international and Chinese scientific journals. Her research interests include software architecture testing and software metrics.

Hui Xu was born in Harbin, Heilongjiang Province, China, in 1984. She is currently a Ph.D. candidate in computer science and technology at Harbin Engineering University, Harbin. She received her B.S. degree in Information and Computer Engineering from Northeast Forestry University, and Master degree in Computer Science and Technology from Harbin Normal University, China, in 2006 and 2009 respectively.

She has published more than 10 papers in international and Chinese scientific journals. Her research interests include social computing and complex networks.