Testing Approach of Component Interaction for Software Architecture

Lijun Lun, Xin Chi, Hui Xu

Abstract—**Components as the smallest granularity are used to composite the software architecture, and they communicate with each other by their interfaces. Effective cooperative work ability is guaranteed by correct component interaction. So, component interaction testing is one of the most important role of software architecture testing when we examine quality of software architecture. Existing researches show that component path coverage is an important test adequacy criterion for software architecture testing. We have presented a set of component path coverage criteria for the test with component interaction relationships. Based on the existing methods of component path coverage and component path-analysis, we propose two component path coverage criteria, component path with node coverage criterion and component path with edge coverage criterion, and discuss the subsumption relationships among these coverage criteria. These coverage criteria define the adequacy of component path test suites at different levels and the test suites satisfying these coverage criteria can detect respectively different types of faults. On this basis, two algorithms are proposed to realize the automatic generation of the corresponding component paths according to two component path coverage criteria, and an experimental method is proposed to analyze the component interaction. The experimental results show that the component interaction for a given component and coverage path coverage criteria is more, the coverage rate of the component is higher, which indicates that its fault-detection capability is stronger.**

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

I. Introduction

SOFTWARE architecture describes the elements of a software system, interoperability of elements, and guides software system pattern and constraint of pattern [1]. It is a bridge between the demand and the realization of the software development process, the explicit description of the overall structure of the software system in abstract layer [2]. In order to describe and design software architecture and verify whether a software architecture satisfies the desired system requirements, the development of an optimal software architecture suitable for system performance requirements is the core problem that must be studied and solved in software architecture development methods [3]. The function of the application system is expressed by the way of the interactions of composed components. Component interaction problem is associated not only with the inner implementation and but also with the closely related to the interactive connection between components. To improve software quality and support software development and software reuse, it is necessary to test the interactive connection between components as early as possible [4].

Component interaction technology is built on message passing principle. The interface is the channel between component and other component. The sender component sends messages through its interfaces, and the receiver component receives the message through its interfaces. A component can directly interact with other components through its interface, or indirectly interact with other components through the interface of intermediate components. Because there may be interaction between components, and the components of system may be used in the system many times. So, modifying a component may have an impact on its related components, even affect the function of the whole system [5]. Therefore, it is necessary to test the component interaction as early as possible to find faults in component interaction [6].

Software architecture testing technology includes the determination test content, the test coverage criteria selection, and the generation of test suites, where the test coverage criteria selection is the central issue in research of software architecture. Traditional software testing can usually be classified into structural coverage [7] and functional coverage [8], Of all the structural coverage criteria, path coverage is relatively rigorous test coverage criterion [9], it requires that enough test suites be designed to ensure that every possible path is executed at least once during testing, so, it has a broader coverage than other coverage criteria.

We have presented a set of component path coverage criteria for C2-style architecture [10], 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 discusses testing approach of component interaction for C2-style architecture. Firstly, set of interactions relationships is defined corresponding to the relationship between component and connector. Then the component interaction graph is constructed to represent static structure of C2-style architecture on the basis of interaction relationships. Based on the component interaction graph introduced, component path with node coverage criterion and component path with edge coverage criterion are proposed to represent test requirements between components, and two algorithms to obtain the component path with node and with edge on coverage criteria are proposed, and an experimental analysis method is proposed. Finally, experimental results and conclusion are given.

II. Related Work

A number of coverage criteria have been proposed in the literature to handle the problem from different aspects. This

Manuscript received November 30, 2017; revised April 19, 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.

section reviews closely related work on path coverage criteria and software architecture coverage criteria.

A. Path Coverage Criteria

McCabe proposed basic path coverage criterion [11]. The criterion considers all possible path sets in a program as a vector space, if there is a group of bases in vector space, then the whole vector space can be covered by the linear combination of the bases. So, we can find out the base in the vector space to test, and if the base doesn't problem, then there isn't problem with all the combinations of paths represented by the base.

Hennell et al. proposed Linear Code Sequence and Jump (LCSAJ) coverage criterion [12]. A LCSAJ is a set of sequentially executed code in a program that it starts the entry point of the first line of program, the branch statement, or the statement that the control flow can jump to arrive. If there exist a number of LCSAJs, and the starting point of first the LCSAJ is the starting point of the program, and the last LCSAJ is the end of the program, then these LCSAJs make up a program path. A LCSAJ path can be composed of multiple LCSAJ.

Miller proposed DD-PATH coverage criterion [13]. The DD-PATH is a chain from a decision node to another decision node in program graph. There is not any internal branch in the chain. The length of a chain represents as the number of edges that the chain contains. Each chain can be divided into a different type of DD-PATH. The DD-PATH is a chain of program graph, and satisfies the following conditions: (1) A single node with an in-degree = 0, (2) A single node with an out-degree = 0, (3) A single node with in-degree $\geq$ 2 or out-degree $\geq$ 2, (4) A single node with in-degree = 1 and out-degree = 1, and (5) The chain is of a maximal length $\geq$ 1.

Zhu proposed elementary path coverage criterion [14]. An elementary path is a path if there are no repeated occurrences of any node. The elementary path coverage criterion requires every feasible complete elementary paths q. there is one path p at least in program graph such that p covers q. In general, an elementary path must be a simple path. The simple path coverage criterion and the elementary path coverage criterion have subsumption relationship. A test suite satisfying the simple path coverage criterion must also satisfy the elementary path coverage criterion, so, the simple path coverage criterion subsumes the elementary path coverage criterion.

Ammann and Offutt proposed simple path coverage criterion [15]. A simple path is a path if no node is visited more than once with the exception that the first node and the last node may be identical. The simple path coverage criterion requires each feasible simple complete path q, there is at least one path p in program graph such that p covers q.

Li et al. proposed R_N(K) path coverage criterion [16]. The coverage criterion obtains the basic path table of program graph through the K value division, which any two paths are independent. The R_N(K) path coverage criterion denoted the set of all paths of length is less than or equal to N in program graph.

For the feasibility of integrity path-tested coverage, Li et al. proposed Length_N path coverage criterion [17]. The coverage criterion conducts the static analysis of the test program, and obtains the branch statement of program which has little affect in program graph, then obtains the corresponding path coverage table according to a certain length of experience. The Length_N path coverage criterion covers the path of length is less than or equal to N of path coverage table.

B. Software Architecture Coverage Criteria

Rosenblum defined two formal adequate test models for component-based software [18]. 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 [19]. In chaining, links represent the dependence relationships that are available 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. proposed a family of architecture-based test criteria based on the chemical abstract machine model [20], such as all-data-elements criterion, all-processing-elements criterion, all-connecting-elements criterion, all-transformations criterion, all-transformation-system criterion, and all-data-dependences criterion, in order to satisfy the requirements of software architecture testing. At the same time, they proposed metrics based on software architecture testability, and used metrics to guide the selection of architecture and the generation of test plans.

Bertolino et al. proposed the formal description of software architecture using chemical abstract machine [21]. From description, they derived Labelled Transition Systems which represented the global system behavior of a concurrent, multi-user software system. They defined the model and test coverage criterion based on chemical abstract machine, derived the integrated test plan of software architecture in order to help guide software architecture testing.

Jin and Offitt [22] 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 [23], 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 [10], 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. Lun et al. proposed basis component path coverage method for software architecture testing [24], and proposed an automatic method to generate basis component paths.

III. C2-Style Architecture Model

In this section, we first introduce the related concepts of the C2-style architecture, and then give the definition of component interaction graph to abstract represent the component interactions. Based on the component interaction graph, we give the definition of 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 [25]; at the same time, the rules of the C2-style are broad enough to render it widely applicable [6].

The C2-style architecture [26] 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 define our intermediate representation Component Interaction Graph (CIG) model and discuss how a C2-style architecture can be represented using our notation [24]. 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 ∪ Conn is the set of nodes, Comp is a finite set of components, each component $Comp_i$ ∈ 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 $Conn_j$ ∈ 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} ⊆ Comp is the set of initial component nodes, these components transmit messages only. That is V_{start} = $\{Comp_i \mid Comp_i.I_{pb\_}i = \emptyset \wedge Comp_i.I_{pb\_}o = \emptyset,\ Comp_i$ ∈ Comp}. V_{end} ⊆ Comp is the set of terminal component nodes, these components receive messages only. That is V_{end} = $\{Comp_i \mid Comp_i.I_{pt\_}o = \emptyset \wedge Comp_i.I_{pt\_}i = \emptyset,\ Comp_i$ ∈ 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.

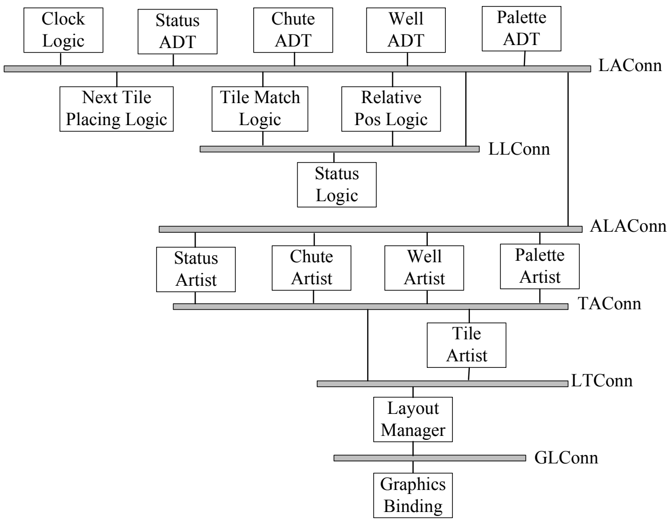

Fig. 1. KLAX Architecture in the C2-Style

In order to construct a representation for the CIG, we conduct the static analysis of the C2-style specification. First, we identify all components and connectors and represent as nodes. Then we 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 $Comp_j$, in such a case, an edge $e \in e_{Conn-Comp}$ is added to connect from the top (bottom) output interface of

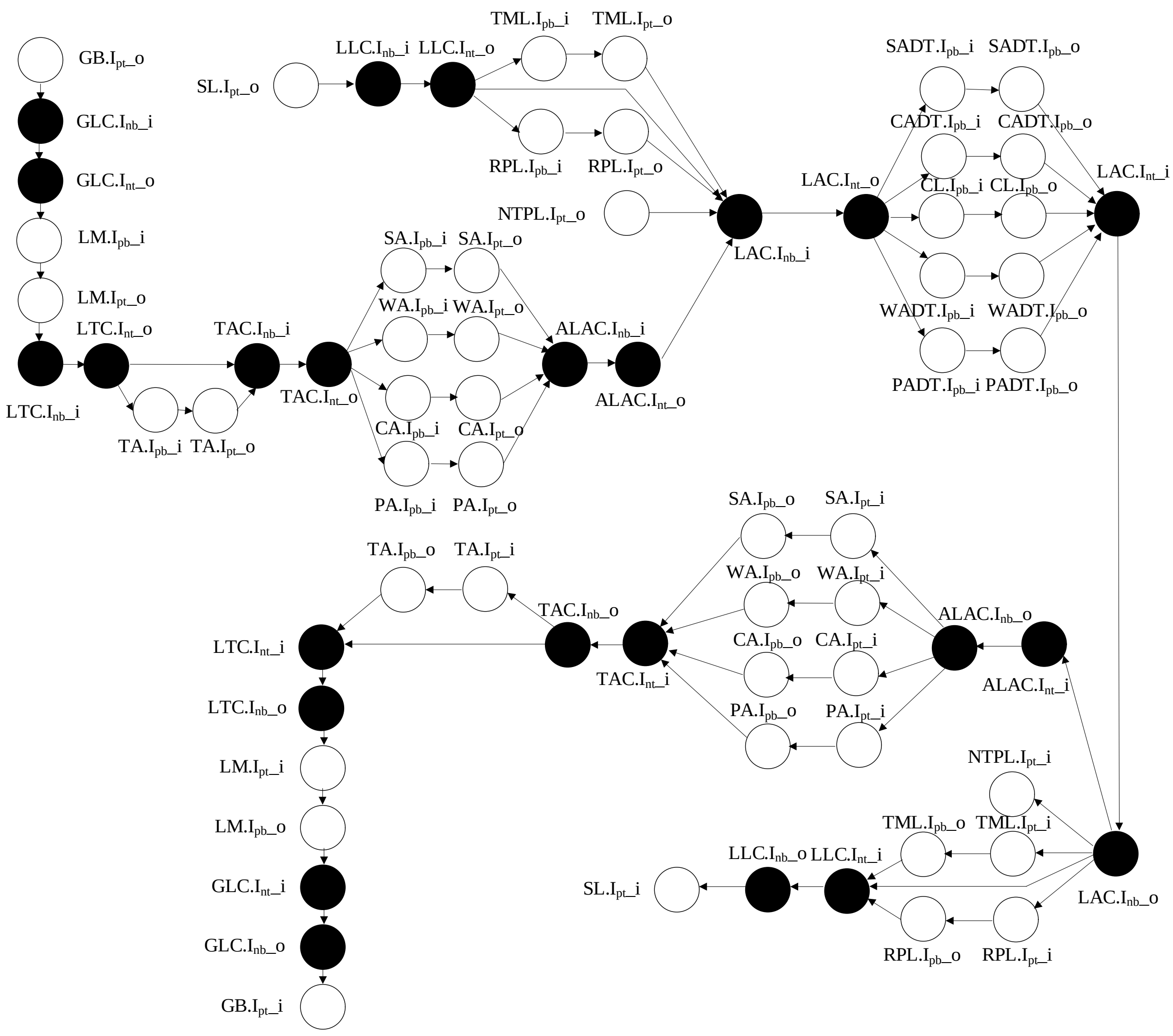

Fig. 2. CIG of KLAX System

$Conn_i$ to the bottom (top) input interface of $Comp_j$ of CIG. If there exists a information flow from connector $Conn_i$ to connector $Conn_j$, in such a case, 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.

In order to illustrate our approach in a better way, we used the well-known KLAX video game application [25]. For this application C2-style architecture has been used. 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, Fig. 2 shows the corresponding CIG for the example KLAX system of Fig. 1 according to C2-style architecture specification [26]. 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 has many new characteristics, such as component, connector and so on, the behavior of each element is part of software architecture, which reflects the interaction between elements. When we describe the software architecture with CIG, the interaction between components can be represented as the component path in the CIG.

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, j = s+1, s+2, ..., t-1, denoted as π_P. 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.

The traditional method to generate test path doesn't suitable for CIG of software architecture. We propose methods to generate component path of CIG.

IV. Component Path Coverage Criteria

In order to ensure the adequacy of testing component path, we must consider the component path coverage criteria of test suites. Component path coverage criteria can measure different test suites in quality and full test; determine the validity of the software architecture testing, decide what time we can stop the software architecture testing, and guide the test suites generation [27]. 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. Component Path with Node Coverage Criterion

Node coverage is one of the most basic coverage method in component path generation. It requires that each node of component path can be covered in software architecture testing.

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, and for any node $C_i \in$ V for i = s+1, s+2, …, t-1, if the π_{CP} covers all nodes and node C_i reachable from C_s to C_t, we call the π_{CP} to satisfy the component path with node coverage criterion, denoted as $CPNCC$.

For example in Fig. 2, we can see that there are two π_{CP}s from component StatusLogic to ChuteADT on $CPNCC$ are shown as follows.

StatusLogic → LLConn → TileMatchLogic → LAConn → ChuteADT

StatusLogic → LLConn → RelativePosLogic → LAConn → ChuteADT

The $CPNCC$ is not complete because it doesn't require all edges from beginning node to stopping node in CIG to be covered, so the fault-detecting ability of $CPNCC$ is limited.

B. Component Path with Edge Coverage Criterion

Edge coverage is strictly stronger than node coverage in component path generation because if we have covered all edges then we have definitely covered all nodes. If we cover all nodes it doesn't necessarily mean that we have covered all edges. It requires that each edge of component path can be covered in software architecture testing.

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, and for any edge $e_{C_i,\ C_j} \in$ E for i = s+1, s+2, …, t-2, j = i+1, i+2, …, t-1, if the π_{CP} covers all nodes and edge $e_{C_i,\ C_j}$ reachable from C_s to C_t, we call the π_{CP} to satisfy the component path with edge coverage criterion, denoted as $CPECC$.

For example in Fig. 2, component path coverage set from component StatusLogic to ChuteADT on $CPNCC$ doesn't cover edge (LLConn, LAConn), so, on the basis of component path coverage set on $CPNCC$, adding a component path can satisfy $CPECC$. Thus, we can see that there are three π_{CP}s from component StatusLogic to ChuteADT on $CPECC$ are shown as follows.

StatusLogic → LLConn → TileMatchLogic → LAConn → ChuteADT

StatusLogic → LLConn → RelativePosLogic → LAConn → ChuteADT

StatusLogic → LLConn → LAConn → ChuteADT

C. Direct Component Path Coverage Criterion

In the process of component interaction, it is necessary to check the messages transmission between components. So, it requires to cover connection of two components through a number of connectors in software architecture testing.

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 each edge $(C_i,\ C_j) \in e_{Conn-Conn}$ for i, j = s+1, s+2, …, t-1, we call the π_{CP} to satisfy the direct component path coverage criterion, denoted as $DCPCC$.

For example in Fig. 2, we can see that there is a π_{CP} from component LayoutManager to StatusArtist on $DCPCC$ is shown as follows.

LayoutManager → LTConn → TAConn → StatusArtist

Meanwhile, there is a π_{CP} from component LayoutManager to GraphicsBinding on $DCPCC$ is shown as follows.

LayoutManager → GLConn → GraphicsBinding

From the definition 5, we can see that there may not exist component path on $DCPCC$ between components. For example in Fig. 2, we can see that there doesn't exist π_{CP} from component LayoutManager to WellADT on $DCPCC$.

D. Indirect Component Path Coverage Criterion

In the process of component interaction, it is necessary to check the messages transmission among components. So, it requires to cover connection of two components through a number of components and connectors in software architecture testing.

Definition 6 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 edge $(C_i,\ C_j)$ E for i, j = s+1, s+2, …, t-1, we call the π_{CP} to satisfy the indirect component path coverage criterion, denoted as $ICPCC$.

From the definition 6, we can see that $ICPCC$ is a more stringent component path coverage criterion, it requires to cover all component paths between two components. Thus, $ICPCC$ has a high practical value.

For example in Fig. 2, we can see that there are two π_{CP}s from component LayoutManager to StatusArtist on $ICPCC$ are shown as follows.

LayoutManager → LTConn → TileArtist → TAConn → StatusArtist

LayoutManager → LTConn → TAConn → StatusArtist

E. Basis Component Path Coverage Criterion

Because $ICPCC$ is a more stringent component path coverage criterion, even if the software architecture is small, there may be a number of components and connectors involved in component interaction, so, there exist a number of component paths and increase the cost of software architecture testing. Therefore, it requires to adopt a simplified method of component path coverage in component path generation.

Definition 7 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 π_{CP} covers all nodes and edges reachable from C_s to C_t, we call the π_{CP} to satisfy the basis component path coverage criterion, denoted as $BCPCC$.

From the definition of $BCPCC$, a component path on $BCPCC$ can be differentiated from all other component paths on $BCPCC$ by at least an edge.

For example in Fig. 2, we can see that there are five π_{CP}s from component LayoutManager to ChuteADT on $BCPCC$ are shown as follows.

LayoutManager $\rightarrow$ LTConn $\rightarrow$ TAConn $\rightarrow$ StatusArtist $\rightarrow$ ALAConn $\rightarrow$ LAConn $\rightarrow$ ChuteADT

LayoutManager $\rightarrow$ LTConn $\rightarrow$ TAConn $\rightarrow$ ChuteArtist $\rightarrow$ ALAConn $\rightarrow$ LAConn $\rightarrow$ ChuteADT

LayoutManager $\rightarrow$ LTConn $\rightarrow$ TAConn $\rightarrow$ WellArtist $\rightarrow$ ALAConn $\rightarrow$ LAConn $\rightarrow$ ChuteADT

LayoutManager $\rightarrow$ LTConn $\rightarrow$ TAConn $\rightarrow$ PaletteArtist $\rightarrow$ ALAConn $\rightarrow$ LAConn $\rightarrow$ ChuteADT

LayoutManager $\rightarrow$ LTConn $\rightarrow$ TileArtist $\rightarrow$ TAConn $\rightarrow$ StatusArtist $\rightarrow$ ALAConn $\rightarrow$ LAConn $\rightarrow$ ChuteADT

However component path π_{CP}: LayoutManager $\rightarrow$ LTConn $\rightarrow$ TileArtist $\rightarrow$ TAConn $\rightarrow$ ChuteArtist $\rightarrow$ ALAConn $\rightarrow$ LAConn $\rightarrow$ ChuteADT doesn't satisfy $BCPCC$, because all edges in this π_{CP} appear in the above five π_{CP}s, this π_{CP} violates the characteristics of the basis component path coverage criterion.

F. Relationships among Component Path Coverage Criteria

Five component path coverage criteria have subsumption relationships among them. According to the definition [28], if any test suite satisfying the coverage criterion C_A satisfies the coverage criterion C_B, then the coverage criterion C_A subsumes the coverage criterion C_B, which can be represented by $C_B \subseteq C_A$. It is obvious that the subsumption relationships between coverage criteria has reflexivity and transitivity, that is, the reflexivity is represented that $C_A \subseteq C_A$, transitivity is represented that if $C_B \subseteq C_A$ and $C_C \subseteq C_B$, then $C_C \subseteq C_A$. Thus, for two component path coverage criteria $CPCC_A$ and $CPCC_B$ and a component path π_{CP_C}, if a triple (π_{CP_C}, TS, $CPCC_A$) can also satisfy another triple (π_{CP_C}, TS, $CPCC_B$) for the same π_{CP_C} and test suite TS, then $CPCC_B \subseteq CPCC_A$.

The test suite TS satisfying the indirect component path coverage criterion also ensures that the CIG satisfies the component path with edge coverage criterion and basis component path coverage criterion. In fact, any test suite satisfying the indirect component path coverage criterion satisfies the component path with edge coverage criterion and basis component path coverage criterion, that is $CPECC \subseteq ICPCC \wedge BCPCC \subseteq ICPCC$. The test suite TS satisfying the component path with edge coverage criterion also ensures that the CIG satisfies the component path with node coverage criterion, and direct component path coverage criterion. In fact, any test suite satisfying the component path with edge coverage criterion satisfies the component path with node coverage criterion and direct component path coverage criterion, that is $CPNCC \subseteq CPECC \wedge DCPCC \subseteq CPECC$. In the same way, $CPNCC \subseteq BCPCC \wedge DCPCC \subseteq BCPCC$. According to the transitivity between coverage criteria, $CPNCC \subseteq ICPCC \wedge DCPCC \subseteq ICPCC$. Because component path coverage set generated on $BCPCC$ is unique, so, it doesn't determine the relationship between $BCPCC$ and $CPECC$.

Thus, we obtain the subsumption relationships among five component path coverage criteria are shown as Fig. 3.

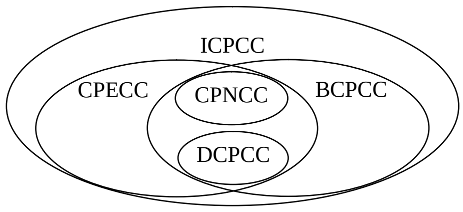

Fig. 3. Subsumption Relationships Among Component Path Coverage Criteria

G. Component Path Coverage Rate

In real applications, when we need to measure the component path coverage criteria on test suits, we need to calculate the component path coverage rate.

Definition 8 A component path coverage criterion be a function CPCC, CPCC: C2 $\times$ C2Spec $\times$ TS $\rightarrow$ [0, 1]. CPCC(C2, C2Spec, TS) = R means that the adequacy of testing of the C2-style architecture C2 by the test suite TS with respect to the specification C2Spec is of coverage rate R according to the component path coverage criterion CPCC. The greater the real number R, the more adequate the testing. R is calculated as follows:

$$R = \frac{||\Pi_{CPCC}||}{||EP(\Pi_{CPCC}(CIG))||} \times 100\% \tag{1}$$

where Π_{CPCC} represents the set of component paths on component path coverage criterion CPCC that covered by test suite TS, $||\Pi_{CPCC}||$ represents the number of elements in Π_{CPCC}, $||EP(\Pi_{CPCC}(CIG))||$ represents the number of component paths on corresponding component path coverage criterion CPCC in CIG.

V. Algorithms of Component Path Generation

In this section, we propose two algorithms to generate component path coverage set from beginning component C_s to stopping component C_t on $CPNCC$ and $CPECC$. Two algorithms contain three procedures as follows:

- Procedure isConnected(C_s, C_t): is used to determine the connectivity of nodes C_s and C_t passing through nodes C_i, C_j, $\ldots$, and 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 Prefix(C_k, π_{Pk}): is used to obtain the prefix of node C_k of component path π_{Pk}.
- Procedure Postfix(C_k, π_{Pk}): is used to obtain the postfix of node C_k of component path π_{Pk}.

A. *Algorithm for determining π_{CP} on $CPNCC$*

Algorithm CPNA can be used to generate the component path coverage set on $CPNCC$. Algorithm CPNA accepts component interaction graph CIG, beginning component C_s and stopping component C_t. The main idea of CPNA algorithm can be briefly stated as follows: Firstly, it determines the connectivity between beginning component C_s and stopping component C_t. Then, it saves nodes into set CPNSet. Finally, it connects nodes of set CPNSet to form the component path coverage set.

Algorithm 1 CPNA(CIG, C_s, C_t, CPNSet)

```
Require: CIG, $C_s$ is beginning component, $C_t$ is stopping component.
Ensure: CPNSet is component path coverage set on $CPNCC$.
Begin
1   if (!isConnected($C_s$, $C_t$)) then
2     return;
3   end if
4   CPNSet = $\emptyset$;
5   for (k1 = s; k1 < t; k1 ++)
6     add $C_{k1}$ to CPNSet;
7     for (k2 = k1 + 1; k2 <= t; k2 ++)
8       if ($e_{C_{k1},\, C_{k2}} \in$ E $\wedge$ $C_{k2} \notin$ CPNSet) then
9         if ($C_{k2}$ = $C_t$) then
10          add $C_{k2}$ to CPNSet;
11          break;
12        else
13          add $C_{k2}$ to CPNSet;
14        end if
15      end if
16    end for
17  end for
18  Output the every node of CPNSet to obtain component path coverage set;
End CPNA
```

We employ the CIG shown in Fig. 2 to demonstrate algorithm CPNA. Let us consider example showing the component path coverage set on $CPNCC$ from component GraphicsBinding to component ClockLogic. That is C_s = GraphicsBinding, C_t = ClockLogic.

Firstly, according to step 1, isConnected(GraphicsBinding, ClockLogic) = true, so, there exists π_{CP} from GraphicsBinding to ClockLogic.

Secondly, according to step 6, CPNSet = {GraphicsBinding}. According to steps 8-14, (GraphicsBinding, GLConn) $\in$ E $\wedge$ GLConn $\notin$ CPNSet, CPNSet = {GraphicsBinding, GLConn}. Repeated steps 8-14, CPNSet = {GraphicsBinding, GLConn, LayoutManager, LTConn, TileArtist, StatusArtist, ChuteArtist, WellArtist, PaletteArtist, ALAConn, LAConn, ClockLogic}.

Thus, according to step 18, CPNSet = {GraphicsBinding $\rightarrow$ GLConn $\rightarrow$ LayoutManager $\rightarrow$ LTConn $\rightarrow$ TileArtist $\rightarrow$ TAConn $\rightarrow$ StatusArtist $\rightarrow$ ALAConn $\rightarrow$ LAConn $\rightarrow$ ClockLogic, GraphicsBinding $\rightarrow$ GLConn $\rightarrow$ LayoutManager $\rightarrow$ LTConn $\rightarrow$ TileArtist $\rightarrow$ TAConn $\rightarrow$ ChuteArtist $\rightarrow$ ALAConn $\rightarrow$ LAConn $\rightarrow$ ClockLogic, GraphicsBinding $\rightarrow$ GLConn $\rightarrow$ LayoutManager $\rightarrow$ LTConn $\rightarrow$ TileArtist $\rightarrow$ TAConn $\rightarrow$ WellArtist $\rightarrow$ ALAConn $\rightarrow$ LAConn $\rightarrow$ ClockLogic, GraphicsBinding $\rightarrow$ GLConn $\rightarrow$ LayoutManager $\rightarrow$ LTConn $\rightarrow$ TileArtist $\rightarrow$ TAConn $\rightarrow$ PaletteArtist $\rightarrow$ ALAConn $\rightarrow$ LAConn $\rightarrow$ ClockLogic}.

As we can see from the above example, the CPNA algorithm can obtain all component paths from beginning component to stopping component on $CPNCC$ when the beginning component and the stopping component are given.

B. *Algorithm for determining π_{CP} on $CPECC$*

Algorithm CPEA can be used to generate the component path coverage set on $CPECC$. Algorithm CPEA accepts component interaction graph CIG, beginning component C_s and stopping component C_t. The main idea of CPEA algorithm can be briefly stated as follows: Firstly, it determines the connectivity of beginning component C_s and stopping component C_t. Then, it calls CPNA algorithm to obtain component path coverage set on $CPNCC$. Finally, it checks all edges $e_{C_{k1},C_{k2}}$ that have not been covered by CPNA algorithm, calls procedure Prefix(C_{k1}, π_{Pk1}) to obtain the prefix of node C_{k1} of π_{Pk1} and calls procedure Postfix(C_{k2}, π_{Pk1}) to obtain the postfix of node C_{k2} of π_{Pk1}, it connects the prefix with C_{k1}, C_{k2}, and the postfix to generate the component path coverage set on $CPECC$.

Algorithm 2 CPEA(CIG, C_s, C_t, CPESet)

```
Require: CIG, $C_s$ is beginning component, $C_t$ is stopping component.
Ensure: CPESet is component path coverage set on $CPECC$.
Begin
1   if (!isConnected($C_s$, $C_t$)) then
2     return;
3   end if
4   CPESet = $\emptyset$;
5   CPNA(CIG, $C_s$, $C_t$, CPNSet);
6   for (k1 = 1; k1 <= |CPNSet|; k1 ++)
7     $\pi_P$ = $\emptyset$;
8     if ($\pi_{Pk1} \in$ CPNSet) then
9       CPESet = CPESet + $\pi_{Pk1}$;
10      for (k2 = 1; k2 <= $|E|$; k2 ++)
11        if ($e_{C_{k1},C_{k2}} \in$ E $\wedge$ $C_{k1} \in \pi_{Pk1}$ $\wedge$ $C_{k2} \in \pi_{Pk1}$ $\wedge$ $e_{C_{k1},C_{k2}} \notin$ CPESet) then
12          TempP1 = Prefix($C_{k1}$, $\pi_{Pk1}$);
13          TpmpP2 = Postfix($C_{k2}$, $\pi_{Pk1}$);
14          $\pi_P$ = TempP1 + $C_{k1}$ + $C_{k2}$ + TempP2;
15        end if
16      end for
17      if ($\pi_P \notin$ CPESet) then
18        CPESet = CPESet + $\pi_P$;
19      end if
20    end if
21  end for
22  return CPESet;
End CPEA
```

We employ the CIG shown in Fig. 2 to demonstrate algorithm CPEA. Let us consider example showing the component path coverage set on $CPECC$ from component WellADT to component LayoutManager. That is C_s = WellADT, C_t = LayoutManager.

Firstly, according to step 1, isConnected(WellADT, LayoutManager) = true, so, there exists π_{CP} from WellADT to LayoutManager.

Secondly, according to step 5, call CPNCC algorithm to obtain component path coverage set CPNSet = {WellADT $\rightarrow$ LAConn $\rightarrow$ ALAConn $\rightarrow$ StatusArtist $\rightarrow$ TAConn $\rightarrow$ TileArtist $\rightarrow$ LTConn $\rightarrow$ LayoutManager, WellADT $\rightarrow$ LAConn $\rightarrow$ ALAConn $\rightarrow$ ChuteArtist $\rightarrow$ TAConn $\rightarrow$ TileArtist $\rightarrow$ LTConn $\rightarrow$ LayoutManager, WellADT $\rightarrow$ LAConn $\rightarrow$ ALAConn $\rightarrow$ WellArtist $\rightarrow$ TAConn $\rightarrow$ TileArtist $\rightarrow$ LTConn $\rightarrow$ LayoutManager, WellADT $\rightarrow$ LAConn $\rightarrow$ ALAConn $\rightarrow$ PaletteArtist $\rightarrow$ TAConn $\rightarrow$ TileArtist $\rightarrow$ LTConn $\rightarrow$ LayoutManager}.

Thirdly, according to step 8, we get π_{P1} = WellADT $\rightarrow$ LAConn $\rightarrow$ ALAConn $\rightarrow$ StatusArtist $\rightarrow$ TAConn $\rightarrow$ TileArtist $\rightarrow$ LTConn $\rightarrow$ LayoutManager. According to step 9, CPESet = {WellADT $\rightarrow$ LAConn $\rightarrow$ ALAConn $\rightarrow$ StatusArtist $\rightarrow$ TAConn $\rightarrow$ TileArtist $\rightarrow$ LTConn $\rightarrow$ LayoutManager}.

Fourth, according to step 11, because $e_{TAConn,LTConn} \in$ E $\wedge$ TAConn $\in \pi_{P1}$ $\wedge$ LTConn $\in \pi_{P1}$ $\wedge$ $e_{TAConn,LTConn}$ $\notin$ CPESet, so, according to steps 12-13, obtains the prefix TempP1 = WellADT $\rightarrow$ LAConn $\rightarrow$ ALAConn $\rightarrow$ StatusArtist of TAConn of π_{P1} and the postfix TempP2 = LayoutManager of LTConn of π_{P1}. Then, according to step 14, connects the prefix with TAConn, LTConn, and the postfix to generate component path π_P = WellADT $\rightarrow$ LAConn $\rightarrow$ ALAConn $\rightarrow$ StatusArtist $\rightarrow$ TAConn $\rightarrow$ LTConn $\rightarrow$ LayoutManager and adds to CPESet. According to step 17, $\pi_P \notin$ CPESet, so, according to step 18, CPESet = {WellADT $\rightarrow$ LAConn $\rightarrow$ ALAConn $\rightarrow$ StatusArtist $\rightarrow$ TAConn $\rightarrow$ TileArtist $\rightarrow$ LTConn $\rightarrow$ LayoutManager, WellADT $\rightarrow$ LAConn $\rightarrow$ ALAConn $\rightarrow$ StatusArtist $\rightarrow$ TAConn $\rightarrow$ LTConn $\rightarrow$ LayoutManager}.

Fifth, according to step 8, we get π_{P2} = WellADT $\rightarrow$ LAConn $\rightarrow$ ALAConn $\rightarrow$ ChuteArtist $\rightarrow$ TAConn $\rightarrow$ TileArtist $\rightarrow$ LTConn $\rightarrow$ LayoutManager. According to step 9, CPESet = {WellADT $\rightarrow$ LAConn $\rightarrow$ ALAConn $\rightarrow$ StatusArtist $\rightarrow$ TAConn $\rightarrow$ TileArtist $\rightarrow$ LTConn $\rightarrow$ LayoutManager, WellADT $\rightarrow$ LAConn $\rightarrow$ ALAConn $\rightarrow$ StatusArtist $\rightarrow$ TAConn $\rightarrow$ LTConn $\rightarrow$ LayoutManager, WellADT $\rightarrow$ LAConn $\rightarrow$ ALAConn $\rightarrow$ ChuteArtist $\rightarrow$ TAConn $\rightarrow$ TileArtist $\rightarrow$ LTConn $\rightarrow$ LayoutManager}. According to step 11, because $e_{TAConn,LTConn} \in$ E $\wedge$ TAConn $\in \pi_{P1}$ $\wedge$ $e_{TAConn,LTConn} \in$ CPESet, CPESet still contains three component paths.

Sixth, according to step 8, we get π_{P3} = WellADT $\rightarrow$ LAConn $\rightarrow$ ALAConn $\rightarrow$ WellArtist $\rightarrow$ TAConn $\rightarrow$ TileArtist $\rightarrow$ LTConn $\rightarrow$ LayoutManager. According to step 9, CPESet = {WellADT $\rightarrow$ LAConn $\rightarrow$ ALAConn $\rightarrow$ StatusArtist $\rightarrow$ TAConn $\rightarrow$ TileArtist $\rightarrow$ LTConn $\rightarrow$ LayoutManager, WellADT $\rightarrow$ LAConn $\rightarrow$ ALAConn $\rightarrow$ StatusArtist $\rightarrow$ TAConn $\rightarrow$ LTConn $\rightarrow$ LayoutManager, WellADT $\rightarrow$ LAConn $\rightarrow$ ALAConn $\rightarrow$ ChuteArtist $\rightarrow$ TAConn $\rightarrow$ TileArtist $\rightarrow$ LTConn $\rightarrow$ LayoutManager, WellADT $\rightarrow$ LAConn $\rightarrow$ ALAConn $\rightarrow$ WellArtist $\rightarrow$ TAConn $\rightarrow$ TileArtist $\rightarrow$ LTConn $\rightarrow$ LayoutManager}. According to step 11, because $e_{TAConn,LTConn} \in$ E $\wedge$ TAConn $\in \pi_{P1}$ $\wedge$ $e_{TAConn,LTConn}$ $\in$ CPESet, CPESet still contains four component paths.

Seventh, according to step 8, we get π_{P4} = WellADT $\rightarrow$ LAConn $\rightarrow$ ALAConn $\rightarrow$ PaletteArtist $\rightarrow$ TAConn $\rightarrow$ TileArtist $\rightarrow$ LTConn $\rightarrow$ LayoutManager. According to step 9, CPESet = {WellADT $\rightarrow$ LAConn $\rightarrow$ ALAConn $\rightarrow$ StatusArtist $\rightarrow$ TAConn $\rightarrow$ TileArtist $\rightarrow$ LTConn $\rightarrow$ LayoutManager, WellADT $\rightarrow$ LAConn $\rightarrow$ ALAConn $\rightarrow$ StatusArtist $\rightarrow$ TAConn $\rightarrow$ LTConn $\rightarrow$ LayoutManager, WellADT $\rightarrow$ LAConn $\rightarrow$ ALAConn $\rightarrow$ ChuteArtist $\rightarrow$ TAConn $\rightarrow$ TileArtist $\rightarrow$ LTConn $\rightarrow$ LayoutManager, WellADT $\rightarrow$ LAConn $\rightarrow$ ALAConn $\rightarrow$ WellArtist $\rightarrow$ TAConn $\rightarrow$ TileArtist $\rightarrow$ LTConn $\rightarrow$ LayoutManager, WellADT $\rightarrow$ LAConn $\rightarrow$ ALAConn $\rightarrow$ PaletteArtist $\rightarrow$ TAConn $\rightarrow$ TileArtist $\rightarrow$ LTConn $\rightarrow$ LayoutManager}. According to step 11, because $e_{TAConn,LTConn} \in$ E $\wedge$ TAConn $\in \pi_{P1}$ $\wedge$ $e_{TAConn,LTConn} \in$ CPESet, CPESet still contains five component paths.

Thus, the component path coverage set on $CPECC$ is CPESet = {WellADT $\rightarrow$ LAConn $\rightarrow$ ALAConn $\rightarrow$ StatusArtist $\rightarrow$ TAConn $\rightarrow$ TileArtist $\rightarrow$ LTConn $\rightarrow$ LayoutManager, WellADT $\rightarrow$ LAConn $\rightarrow$ ALAConn $\rightarrow$ StatusArtist $\rightarrow$ TAConn $\rightarrow$ LTConn $\rightarrow$ LayoutManager, WellADT $\rightarrow$ LAConn $\rightarrow$ ALAConn $\rightarrow$ ChuteArtist $\rightarrow$ TAConn $\rightarrow$ TileArtist $\rightarrow$ LTConn $\rightarrow$ LayoutManager, WellADT $\rightarrow$ LAConn $\rightarrow$ ALAConn $\rightarrow$ WellArtist $\rightarrow$ TAConn $\rightarrow$ TileArtist $\rightarrow$ LTConn $\rightarrow$ LayoutManager, WellADT $\rightarrow$ LAConn $\rightarrow$ ALAConn $\rightarrow$ PaletteArtist $\rightarrow$ TAConn $\rightarrow$ TileArtist $\rightarrow$ LTConn $\rightarrow$ LayoutManager}.

As we can see from the above example, the CPEA algorithm can obtain all component paths from beginning component to stopping component on $CPECC$ when the beginning component and the stopping component are given.

VI. Experimental Studies

In this section, we take KLAX system as an example and apply the proposed component path coverage criteria and algorithms to investigate the effectiveness and performance of our proposed component path coverage criteria, and propose an analysis of the experimental results.

A. Experimental Results

We use KLAX system to perform the experiments. To simplify the experimental results, we statistic the component path coverage set for component StatusLogic on $CPNCC$ and $CPECC$ are shown in Table I. In the table, the first column represents the component paths for component StatusLogic on $CPNCC$, the second column represents the component paths for component StatusLogic on $CPECC$.

From Table I, it is can be seen that the number of component paths for component StatusLogic on $CPNCC$ is 12, that is $||\Pi_{CPNCC}||$ = 12. By calculation, the total number of component paths for all components on $CPNCC$ in CIG is 244, that is $||EP(\Pi_{CPNCC}(CIG))||$ = 244. So, according to Equation (1), the coverage rate for component StatusLogic on $CPNCC$ is R_{CPNCC} = 12 / 244 $\times$ 100% = 4.92%. The number of component paths for component StatusLogic on $CPECC$ is 17, that is $||\Pi_{CPECC}||$ = 17. By calculation, the total number of component paths for all components on $CPECC$ in CIG is 290, that is $||EP(\Pi_{CPECC}(CIG))||$ = 290. So, according to Equation (1), the coverage rate for component StatusLogic on $CPECC$ is R_{CPECC} = 17 / 290 $\times$ 100% = 5.86%.

TABLE I
COMPONENT PATH COVERAGE SET FOR COMPONENT STATUSLOGIC ON CPNCC AND CPECC

CPNCC	CPECC
StatusLogic→LLConn→TileMatchLogic	StatusLogic→LLConn→TileMatchLogic
StatusLogic→LLConn→TileMatchLogic→LAConn→ClockLogic	StatusLogic→LLConn→TileMatchLogic→LAConn→ClockLogic
StatusLogic→LLConn→TileMatchLogic→LAConn→StatusADT	StatusLogic→LLConn→TileMatchLogic→LAConn→StatusADT
StatusLogic→LLConn→TileMatchLogic→LAConn→ChuteADT	StatusLogic→LLConn→TileMatchLogic→LAConn→ChuteADT
StatusLogic→LLConn→TileMatchLogic→LAConn→WellADT	StatusLogic→LLConn→TileMatchLogic→LAConn→WellADT
StatusLogic→LLConn→TileMatchLogic→LAConn→PaletteADT	StatusLogic→LLConn→TileMatchLogic→LAConn→PaletteADT
StatusLogic→LLConn→RelativePosLogic	StatusLogic→LLConn→RelativePosLogic
StatusLogic→LLConn→RelativePosLogic→LAConn→ClockLogic	StatusLogic→LLConn→RelativePosLogic→LAConn→ClockLogic
StatusLogic→LLConn→RelativePosLogic→LAConn→StatusADT	StatusLogic→LLConn→RelativePosLogic→LAConn→StatusADT
StatusLogic→LLConn→RelativePosLogic→LAConn→ChuteADT	StatusLogic→LLConn→RelativePosLogic→LAConn→ChuteADT
StatusLogic→LLConn→RelativePosLogic→LAConn→WellADT	StatusLogic→LLConn→RelativePosLogic→LAConn→WellADT
StatusLogic→LLConn→RelativePosLogic→LAConn→PaletteADT	StatusLogic→LLConn→RelativePosLogic→LAConn→PaletteADT
	StatusLogic→LLConn→LAConn→ClockLogic
	StatusLogic→LLConn→LAConn→StatusADT
	StatusLogic→LLConn→LAConn→ChuteADT
	StatusLogic→LLConn→LAConn→WellADT
	StatusLogic→LLConn→LAConn→PaletteADT

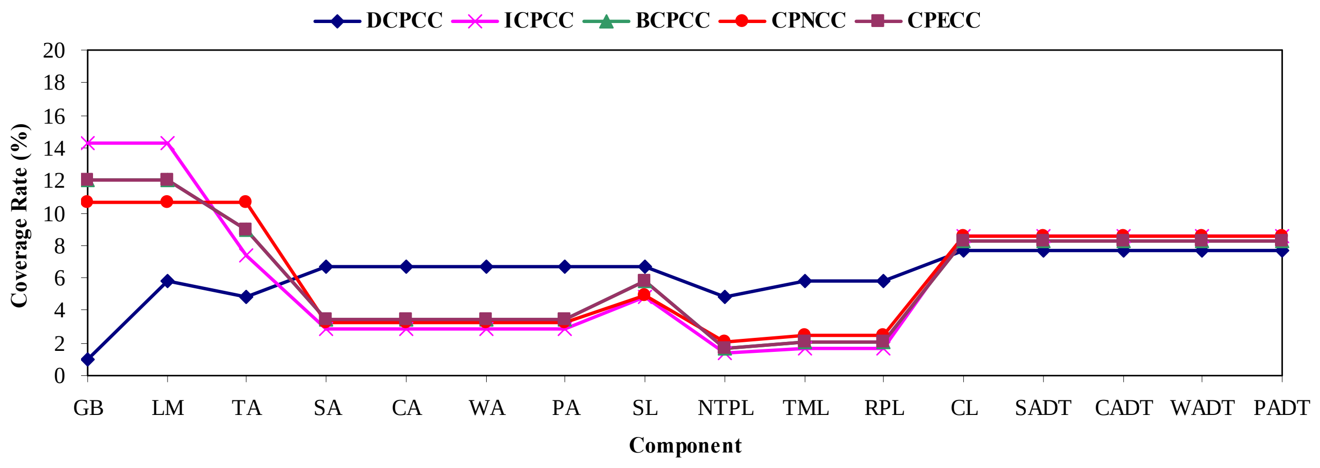

Fig. 4. Coverage Rate for Different Component Path Coverage Criteria on KLAX System

B. Experimental Results Analysis

We evaluate our approaches by five component path coverage criteria case studies, and calculate the coverage rate of all components. The range of coverage rate is given in Fig. 4. From the experimental results of coverage rate, we can clearly see that the component path coverage rate for all components on $DCPCC$ is between 0.96% and 7.69%, the component path coverage rate for all components on $ICPCC$ is between 1.43% and 14.29%, the component path coverage rate for all components on $BCPCC$ is between 1.72% and 12.07%, the component path coverage rate for all components on $CPNCC$ is between 2.05% and 10.66%, and the component path coverage rate for all components on $CPECC$ is between 1.72% and 12.07%.

Summary statistics based on the coverage rate for five component path coverage criteria are given in Table II. In the table, the first column represents the component of KLAX system, the second to sixth columns represent the mean, the median, the standard deviation, 95% confidence interval, and 98% confidence interval of coverage rate on five component path coverage criteria.

The means represent the average value of coverage rate for each component on five component path coverage criteria. From Table II, we can see that the mean of component NextTilePlacingLogic is the smallest, indicating that the number of components that interact with NextTilePlacingLogic is the least. Meanwhile, the mean of component LayoutManager is the largest, indicating that the number of components that interact with LayoutManager is the most.

The medians represent the dividing line of coverage rate for each component on five component path coverage criteria. From Table II, we can see that the median of component NextTilePlacingLogic is the smallest, and it also indicates that the number of components that interact with NextTilePlacingLogic is the least. Meanwhile, the median of component GraphicsBinding and LayoutManager is the largest, indicating that the number of components that interact with GraphicsBinding and LayoutManager is the most.

The standard deviations illustrate the dispersion of component rate for each component on five component path

TABLE II
DESCRIPTIVE STATISTICS OF THE ANALYZED COMPONENT PATH COVERAGE RATE

Component name	R_{CPCC} (%)				
	Mean	Median	Standard deviation	95% Confidence interval	98% Confidence interval
GraphicsBinding	10.4540	12.0700	5.52765	[3.5905, 17.3175]	[1.1914, 19.7166]
LayoutManager	11.4160	12.0700	3.51413	[7.0526, 15.7794]	[5.5274, 17.3046]
TileArtist	7.8600	7.4300	2.16474	[5.1721, 10.5479]	[4.2326, 11.4874]
StatusArtist	3.8360	3.2800	1.63848	[1.8016, 5.8704]	[1.0904, 6.5816]
ChuteArtist	3.8360	3.2800	1.63848	[1.8016, 5.8704]	[1.0904, 6.5816]
WellArtist	3.8360	3.2800	1.63848	[1.8016, 5.8704]	[1.0904, 6.5816]
PaletteArtist	3.8360	3.2800	1.63848	[1.8016, 5.8704]	[1.0904, 6.5816]
StatusLogic	5.4460	4.9200	0.83419	[4.4102, 6.4818]	[4.0482, 6.8438]
NextTilePlacingLogic	2.2880	1.7200	1.43284	[0.5089, 4.0671]	[-0.1130, 4.6890]
TileMatchLogic	2.7440	2.0700	1.71973	[0.6087, 4.8793]	[-0.1377, 5.6257]
RelativePosLogic	2.7440	2.0700	1.71973	[0.6087, 4.8793]	[-0.1377, 5.6257]
ClockLogic	8.4340	8.5700	0.38882	[7.8612, 8.8268]	[7.6925, 8.9955]
StatusADT	8.4340	8.5700	0.38882	[7.8612, 8.8268]	[7.6925, 8.9955]
ChuteADT	8.4340	8.5700	0.38882	[7.8612, 8.8268]	[7.6925, 8.9955]
WellADT	8.4340	8.5700	0.38882	[7.8612, 8.8268]	[7.6925, 8.9955]
PaletteADT	8.4340	8.5700	0.38882	[7.8612, 8.8268]	[7.6925, 8.9955]

coverage criteria. From Table II, we can see that the standard deviation of component GraphicsBinding is the highest, indicating that there is distinct difference between most coverage rate and its mean. The standard deviation of component ClockLogic, StatusADT, ChuteADT, WellADT, and PaletteADT is the smallest, indicating that coverage rate is close to mean. The standard deviation of the component StatusArtist, RelativePosLogic and so on is less than the standard deviation of component GraphicsBinding and is larger than the standard deviation of ClockLogic and so on, indicating that there is a small difference between component coverage and its mean. Thus, the gap of coverage rate of component GraphicsBinding on five component path coverage criteria is much larger than the gap of coverage rate of component ClockLogic, StatusADT, ChuteADT, WellADT, and PaletteADT on five component path coverage criteria.

The confidence intervals illustrate the precision with which we are able to report the effect data. In Table II for five component path coverage criteria, if the confidence interval for mean is referred to as 95%, then the average confidence for mean for the [4.2790, 8.2230]. If the confidence interval for mean is referred to as 98%, then the average confidence interval for mean for the [3.5897, 8.9123].

From the experimental results, it can be observed that the confidence interval for mean from 95% to 98% for component path coverage criteria, the average confidence interval for mean increases from 3.9440 to 5.3226, increasing the proportion of 1.3786%. The length of the confidence interval is the smaller the better, because the length of the confidence interval reflects the degree of precision of the parameter estimates.

VII. CONCLUSION

We have presented a set of component path coverage criteria in software architecture testing. For test coverage, we extend component path coverage criteria in order to capture the component interactions information of software architecture, that is component path with node coverage criterion and component path with edge coverage criterion. We then propose two algorithms to generate the component path coverage set using corresponding component path coverage criteria. The component path coverage sets that satisfy these component path coverage criteria can detect respectively different types of faults. This paper provides a method to analyze experimental results of coverage rate for five component path coverage criteria. The experimental results show the component interaction for a given component is more, the coverage rate of the component is higher, its fault-detection capability is stronger. For component in the middle level, because of its interactive component is less, so the coverage rate is low, the fault-detection ability is lower than other components. At the same time, we will examine the adequacy of these component path 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] M. Shaw and D. Garlan, "Software architecture: perspectives on an emerging discipline," Prentice Hall, 1996.

[2] L. Bass, P. Clements and R. Kazman, "Software architecture in practice," Addison Wesley, 2003.

[3] A. Bertolino, P. Inverardi and H. Muccini, "Software architecture-based analysis and testing: a look into achievements and future challenges," *Computing*, vol. 95, no. 8, pp 633-648, 2013.

[4] 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.

[5] 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.

[6] 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.

[7] M. Alshraideh, L. Bottaci and A. B. Mahafzah, "Using program data-state scarcity to guide automatic test data generation," *Software Quality Control*, vol. 18, no. 1, pp. 109-144, 2010.

[8] W. L. Andrade and P. D. L. Machado, “Generating test cases for real-time systems based on symbolic models,” *IEEE Trans. Softw. Engi.*, vol. 39, no. 9, pp. 1216-1229, 2013.
[9] A. Bertolino M. Marré, “How many paths are needed for branch testing,” *Journal Systems and Software*, vol, 35, no. 2, pp. 95-106, 1996.
[10] 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.
[11] T. J. McCabe, “A complexity measure,” *IEEE Trans. Softw. Engi.*, vol. 2, no. 4, pp. 308-320, 1976.
[12] M. A. Hennell, M. R. Woodward, D. Hedley, “On program analysis,” *Information Processing Letters*, vol. 5, no. 5, pp. 136-140, 1976.
[13] E. F. Miller, “Tutorial: program testing techniques,” in *Proceedings of IEEE Annual Computer Software and Applications Conference*, November 1977, pp. 107-120.
[14] Zhu, “Axiomatic assessment of control flow-based software test adequacy criteria,” *Software Engineering Journal*, vol. 10, no. 5, pp. 194-204, 1995.
[15] P. Ammann and J. Offutt, “Introduction to software testing,” Cambridge University Press, 2008.
[16] B. L. Li, Z. S. Li, H. Jin, J. R. Sun and Y. H. Chen, “Test case generation base on R_N(K) criterion annealing algorithm,” *Journal of Jilin University (Engineering and Technology Edition)*, vol. 38, no. 3, pp. 680-684, 2008,
[17] 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.
[18] D. S. Rosenblum, “Adequate testing of component-based software,” Technical Report, TR97-34, 1997.
[19] J. A. Stafford, D. J. Richardson and A. L. Wolf, “Chaining: a software architecture dependence analysis technique,” Technical Report CU-CS-845-97, 1997.
[20] D. J. Richardson, J. A. Stafford and A. L. Wolf, “A formal approach to architecture-based software testing,” Technical Report, 1998.
[21] A. Bertolino, F. Corradini, P. Inverardi and H. Muccini, “Deriving test plans from architectural description,” in *Proceedings of the International Conference on Software Engineering*, June 2000, pp. 220-229.
[22] 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.
[23] 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.
[24] 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.
[25] 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.
[26] 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.
[27] 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.
[28] 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.

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. Currently, 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. Currently, 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. Currently, her research interests include social computing and complex networks.