Component Dependency Path Coverage Criteria for C2-Style Architecture Testing

Lijun Lun and Xin Chi

***Abstract*—A number of coverage approaches have been proposed for testing software architecture. Previous researches mainly focused on the dependency relationships between component and connector, they rarely involved the dependency relationships between components. We investigate how interaction information between components can be used to improve software architecture testing. Given a C2-style architecture specified using C2 architecture description language (C2-ADL), we first construct an intermediate representation, named the component dependency graph (G_{CD}). G_{CD} combines information extracted components, connectors, and dependency relationships between component and connector, and propose three component dependency path coverage criteria based on the G_{CD}, direct component dependency path coverage criterion, indirect component dependency path coverage criterion, and Length-N component dependency path coverage criterion. Algorithms are presented to compute the component dependency path coverage rate on three component dependency path coverage criteria. It reports on typical C2-style architecture studies whose experimental results show that the direct and indirect component dependency path coverage rate increases from 39.41% to 57.14% for top/bottom components. The component dependency path of length n coverage rate decreases from 100% to 32% for top/bottom components. However, the component dependency path of length 2, 3, 4, 5, and 6 coverage rate increases from 8% to 37.32% for middle levels components. The assessment of test coverage criteria will provide software testers with a guide to apply each component dependency path coverage criteria.**

***Index Terms*—C2-style architecture testing, component dependency graph, component dependency path, coverage criterion, coverage rate.**

I. Introduction

SOFTWARE architecture is set of components, connectors, and constraints and configurations [1], the corresponding development method takes software architecture as the core artifact of the software design phase, and becomes an important means to control the complexity of software systems, to improve software quality and to support software development and software reuse. The purpose of software architecture analysis and evaluation is to identify the potential risks and help make proper architecture decision. In software architecture, component path testing is an important part in software architecture testing [2]. An appropriate component path for software architecture testing can reduce test cost. Different component testing path, need different test cost. So, to reduce the test cost is important target to determine the component path testing as soon as possible.

Manuscript received April 18, 2015; revised August 6, 2015. 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 a Professor, College of Computer Science and Information Engineering, Harbin Normal University, Harbin, China. E-mail:lunlijun@yeah.net.

Xin Chi is with the College of Computer Science and Information Engineering, Harbin Normal University, Harbin, China. E-mail:xinc1990@163.com.

Current software architecture testing research divided into two categories [3]. One is to improve the traditional software testing techniques and methods, so that they service for software architecture testing. The other is to develop new software architecture testing techniques and methods, so that it can better solve problems of software architecture testing. Formalization testing based on software architecture has improved the quality of the software products. Automatic test coverage generation [4] is a hotspot and difficulty in the field of software architecture testing.

Dependency is a use relationship, which represents a thing specification changes may affect the change used by other things. In the software architecture, components communicate with each other to share information, and to provide a system function [5]. Components need to conform a component model in order to allow them to be independently deployed and composed as is i.e. achieve the purpose of their creation [6]. Component dependency is caused by an interaction of components in software architecture. Software architecture functions provided by multiple components, thus modifying a component may affect the function of the whole system [7].

This paper proposes three path coverage criteria to analyze component dependencies for C2-style architecture. Dependency represents the relationships between component and connector that exist in C2-style architecture specification. Firstly, set of dependency relationships is defined corresponding to the relationship between component and connector. Then the component dependency graph of C2-style architecture is constructed on the basis of these dependency relationships. Based on the component dependency graph introduced, direct component dependency path coverage criterion, indirect component dependency path coverage criterion, and Length-N component dependency path coverage criterion are proposed. And present algorithms to compute the component dependency coverage rate on component dependency coverage criteria. And finally, experimental results and conclusion are given.

II. Related work

The paths that will to be tested must be generated or determined before path testing. In this section, we review some kinds of path generating methods and software architecture coverage methods.

Marré and Bertolino introduced the spanning sets of entities for a coverage criterion which can help reduce the cost of testing without impairing testing efficacy [8]. Costa and Monteiro presented an observability coverage-directed vector generation method [9]. This method addressed the problem of

finding a minimal set of execution paths that achieve a user-specified level of observability coverage. This method used Pseudo-Boolean Optimization (PBO) to model the problem of finding the paths that are most likely to increase code coverage. Generated paths are then validated to check for feasibility.

Path coverage criterion requires that all of execution paths are executed during testing. In large-scale software, the existing of infeasible paths in program increases assumptions of software testing and impacts accuracy of testing seriously. Therefore, some simplified path testing techniques are often used in practical application [10]. Li et al. proposed a Length_N path coverage criterion [11]. 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.

Practices show that most of the faults are generated in interaction between components. Brim et al. proposed a component-interaction automata to component interaction specification and verification process [12]. The model is designed to preserve all the interaction properties to provide a rich base for further verification, and allows the system behavior to be configurable according to bindings among components and other specifics. Wu et al. presented a test model that depicts a generic infrastructure of component-based systems and suggests key test elements [13]. The test model is realized using a component interaction graph (CIG) in which the interactions and the dependence relationships among components are illustrated. By utilizing the CIG, they proposed a family of test adequacy criteria which allow optimization of the balancing among budget, schedule, and quality requirements typically necessary in software development.

Li proposed a matrix-based approach to analyzing dependencies in CBSs [14]. To make it possible, this approach identified four types of dependencies in a CBS, and then presented a dependency-based representation called the component dependency graph (CDG) and the dependency matrix (DM) to explicitly represent these dependencies in a CBS. Based on the CDG of a CBS, they proposed eight types of dependency-based test coverage criteria for the system, and built a mathematical basis for managing and analyzing dependencies in a CBS.

Yoon et al. developed RACSET, a process and infrastructure for testing component-based systems [15]. At a high level the process has several parts. First, developers model the system under test using a formal representation with two parts: a directed acyclic graph called the Component Dependency Graph (CDG) and a set of Annotations. A CDG specifies the components making up the system and specifies inter-component dependencies by connecting components with AND and XOR relationships. Annotations include version identifiers for components, and constraints between components or over configurations, written in first-order logic. They formally model the configuration space for component-based systems and use the model to generate test plans covering user-specified portion of the space - the example is covering all direct dependencies between components.

Jin and Offutt proposed a technology of generating test cases [16] in view of architecture description language Wright, according to Interface Connectivity Graph (ICG) and Behavior Graph (BG), and developed testing criteria for generating software architecture level tests from software architecture descriptions.

Gao et al. focused on component test coverage issues, and proposed test models (CFAGs and D-CFAGs) [17] to represent a component's API-based function access patterns in static and dynamic views. A set of component API-based test criteria is defined based on the test models, and a dynamic test coverage analysis approach is provided.

Hashim et al. presented Connector-based Integration Testing for Component-based Systems (CITECB) with an architectural test coverage criteria [18], and describe the test models used that are based on probabilistic deterministic finite automata which are used to represent gate usage profiles at run-time and test execution. It also provides a measuring mechanism of how well the existing test suite are covering the component interactions and provides a test suite coverage monitoring mechanism to reveal the test elements that are not yet covered by the test suites.

Muccini et al. proposed a specialization and refinement of your general approach for software architecture based conformance testing [19], [20], he dealt with the software architecture to code mapping rules imposed by the C2 framework helps to limit the mapping problem, and allows a systematic testing approach.

Lun et al. presented a dependency coverage method for C2-style architecture [5], [21], which included dependency edge coverage, direct component dependency coverage, and indirect component dependency coverage, it covered all the testing component and connector, and reduced scale of testing coverage set, so that test the C2-style architecture effectively.

III. Dependency relationships for C2-style

The C2 architecture style as a kind of classical architecture style, which supports large-grained software reuse and flexible system assembly, and demonstrated in numerous applications across several domains [22]; at the same time, the rules of the C2-style are broad enough to render it widely applicable.

A. *Representation for C2-style architecture*

The C2-style architecture is primarily concerned with high-level system composition issues [19]. It consists of components and connectors, which transmit messages between components. Components maintain states, perform operations, and exchange messages with other components via two interfaces (named top and bottom). Each interface consists of a set of messages that may be sent or received. Inter-component messages are classified into two types, viz. requests to a component to perform an operation, and notifications that a given component has performed an operation or changed state. In the C2-style architecture, both components and connectors have a top and a bottom interface. Systems are composed in a layered style, where a component's top interface may be connected to the bottom interface of a connector, and its bottom interface may be connected to the top interface of another connector. Each connector's

TABLE I
DEPENDENCY RELATIONSHIPS IN C2-STYLE ARCHITECTURE

Dependency relationship	Notation	Description
between interface of component and connector	$DEP_{pn}(Comp_1.I_p^t_o, Conn_2.I_n^b_i)$ $DEP_{pn}(Comp_1.I_p^b_o, Conn_2.I_n^t_i)$ $DEP_{np}(Comp_1.I_p^b_i, Conn_2.I_n^t_o)$ $DEP_{np}(Comp_1.I_p^t_i, Conn_2.I_n^b_o)$ $DEP_{nn}(Conn_1.I_{n_1}^t_o, Conn_2.I_{n_2}^b_i)$ $DEP_{nn}(Conn_1.I_{n_1}^b_o, Conn_2.I_{n_2}^t_i)$	change of $Comp_1.I_p^t_o$ affects $Conn_2.I_n^b_i$ change of $Comp_1.I_p^b_o$ affects $Conn_2.I_n^t_i$ change of $Conn_2.I_n^t_o$ affects $Comp_1.I_p^b_i$ change of $Conn_2.I_n^b_o$ affects $Comp_1.I_p^t_i$ change of $Conn_1.I_{n_1}^t_o$ affects $Conn_2.I_{n_2}^b_i$ change of $Conn_1.I_{n_1}^b_o$ affects $Conn_2.I_{n_2}^t_i$
between component and connector	$DEP_{pn}(Comp_1, Conn_2)$ $DEP_{np}(Comp_1, Conn_2)$ $DEP_{nn}(Conn_1, Conn_2)$	$DEP_{pn}(Comp_1.I_p^t_o, Conn_2.I_n^b_i) \vee$ $DEP_{pn}(Comp_1.I_p^b_o, Conn_2.I_n^t_i)$ $DEP_{np}(Comp_1.I_p^b_i, Conn_2.I_n^t_o) \vee$ $DEP_{np}(Comp_1.I_p^t_i, Conn_2.I_n^b_o)$ $DEP_{nn}(Conn_1.I_{n_1}^t_o, Conn_2.I_{n_2}^b_i) \vee$ $DEP_{nn}(Conn_2.I_{n_2}^b_o, Conn_1.I_{n_1}^t_i)$
in component and connector	$DEP_p(Comp_1)$ $DEP_n(Conn_2)$	a bottom of $Comp_1$ depends on a top of $Comp_1$, or a top of $Comp_1$ depends on a bottom of $Comp_1$ a bottom of $Conn_2$ depends on a top of $Conn_2$, or a top of $Conn_2$ depends on a bottom of $Conn_2$

interface may be connected to any number of components or connectors.

We represent a component as $Comp_i$, where $Comp_i \in Comp$ indicates i^{th} component of C2-style architecture and $Comp$ represents a set of components. I_p^t is top interface finite set of $Comp_i$, it consists of the set of top output interfaces $I_p^t_o$ and top input interface $I_p^t_i$. I_p^b is bottom interface finite set of $Comp_i$, it consists of the set of bottom output interfaces $I_p^b_o$ and bottom input interface $I_p^b_i$. $Comp_i$ uses $I_p^t_o$ or $I_p^b_o$ to send an event request, and uses $I_p^t_i$ or $I_p^b_i$ to receive other component or connector send messages. If $Comp_i$ doesn't the top or bottom output interface, then $Comp_i$ can't send messages from it's top or bottom output interface. Similarly, if $Comp_i$ doesn't the top or bottom input interface, then $Comp_i$ can't receive messages from it's top or bottom input interface.

We represent a connector as $Conn_j$, where $Conn_j \in Conn$ indicates j^{th} connector of C2-style architecture and $Conn$ represents a set of connectors. I_n^t is top interface finite set of $Conn_j$, it consists of the set of top output interfaces $I_n^t_o$ and top input interface $I_n^t_i$. I_n^b is bottom interface finite set of $Conn_j$, it consists of the set of bottom output interfaces $I_n^b_o$ and bottom input interface $I_n^b_i$. $Conn_j$ uses $I_n^t_o$ or $I_n^b_o$ to send an event request, and uses $I_n^t_i$ or $I_n^b_i$ to receive other component or connector send messages.

We represent a constraint as $(Comp_i.I_p^t_o \rightarrow Conn_j.I_n^b_i) \cup (Comp_i.I_p^b_o \rightarrow Conn_j.I_n^t_i) \cup (Conn_i.I_n^t_o \rightarrow Comp_j.I_p^b_i) \cup (Conn_i.I_n^b_o \rightarrow Comp_j.I_p^t_i) \cup (Conn_i.I_n^t_o \rightarrow Conn_j.I_n^b_i) \cup (Conn_i.I_n^b_o \rightarrow Conn_j.I_n^t_i)$. It means that there exists a path from the output interface of component (connector) to the input interface of connector (component), or the output interface of connector to the input interface of connector.

B. Dependency relationships in the C2-style

Dependency relationships [5] the architectural level arise from the connections between component, connector, and constraint on their interactions. These relationships may involve some form of control or data flow, but more generally involve source structure and behavior. Source structure (or structure, for short) has to do with static source specification dependencies, while behavior has to do with dynamic interaction dependencies.

Table I provides a summary of these dependency relationships in C2-style architecture.

Here are the descriptions of these dependency relationships, where $Comp_i \in$ Comp represents a component of Comp, I_p^t and I_p^b are the set of top interfaces and the set of bottom interfaces of $Comp_i$, $Conn_j \in$ Conn represents a connector of Conn, I_n^t and I_n^b are the set of top interfaces and the set of bottom interfaces of $Conn_j$.

$DEP_{pn}(Comp_1.I_p^t_o, Conn_2.I_n^b_i)$ and $DEP_{pn}(Comp_1.I_p^b_o, Conn_2.I_n^t_i)$ can be used to represent dependency relationship between a interface of component $Comp_1$ and a interface of connector $Conn_2$ in the description. Similarly, $DEP_{np}(Comp_1.I_p^b_i, Conn_2.I_n^t_o)$ and $DEP_{np}(Comp_1.I_p^t_i, Conn_2.I_n^b_o)$ can be used to represent dependency relationship between a interface of connector $Conn_2$ and a interface of component $Comp_1$. $DEP_{nn}(Conn_1.I_{n_1}^t_o, Conn_2.I_{n_2}^b_i)$ and $DEP_{nn}(Conn_1.I_{n_1}^b_o, Conn_2.I_{n_2}^t_i)$ can be used to represent dependency relationship between a interface of connector $Conn_1$ and a interface of connector $Conn_2$.

$DEP_{pn}(Comp_1, Conn_2)$ can be used to represent dependency relationship between component $Comp_1$ and connector $Conn_2$ in the description. Similarly, $DEP_{np}(Comp_1, Conn_2)$ can be used to represent dependency relationship between connector $Conn_2$ and component $Comp_1$. $DEP_{nn}(Conn_1, Conn_2)$ can be used to represent dependency relationship between connector $Conn_1$ and connector $Conn_2$.

$DEP_p(Comp_1)$ can be used to represent dependency relationship between two interfaces within component $Comp_1$ in the description. Similarly, $DEP_n(Conn_2)$ can be used to represent dependency relationship between two interfaces within connector $Conn_2$.

IV. Graph Representation Using Dependency Relationships

In order to test dependency relationships between components for C2-style architecture. In this section, we present definition of our representation based on dependency relationships for C2-style architecture, named Component Dependency Graph (G_{CD}), and component dependency path based on G_{CD}.

A. Component dependency graph

As shown in [5], the component dependency graph is a digraph whose node represents component or connector, and edge represents possible information flows between a component and a connector in the C2-ADL specification.

Definition 1 (Component dependency graph). Given a C2-style architecture, the component dependency graph is denoted by G_{CD} = (N, E), where N = Comp ∪ Conn is the set of components and connectors. E is the set of dependency edges between component and connector. A G_{CD} may contain the following three types of dependency edges:

- if $(C_i,\ C_j) \in DEP_{pn}(Comp, Conn)$, it represents an edge called a dependency edge from component to connector, called $DEComp - Conn$ for short.
- if $(C_j,\ C_i) \in DEP_{np}(Comp, Conn)$, it represents an edge called a dependency edge from connector to component, called $DEConn - Comp$ for short.
- if $(C_i,\ C_j) \in DEP_{nn}(Conn, Conn)$, it represents an edge called a dependency edge from connector to connector, called $DEConn - Conn$ for short.

The G_{CD} is constructed from a static analysis of C2-style architecture specification, and outlines the following five steps.

- For each component and connector of architecture description, the corresponding node is added to the G_{CD}.
- For each interface of component and connector, the corresponding node is attached to the component and connector.
- Add the dependency edge from component to connector to attach the G_{CD}.
- Add the dependency edge from connector to component to attach the G_{CD}.
- Add the dependency edge from connector to connector to attach the G_{CD}.

Obviously, these types of dependency edges belong to the set of E.

We introduce a video game KLAX system [23] to illustrate the proposed notions. The design of the system is given in Fig. 1.

The components of the KLAX game can be divided into three logical groups. At the top of the architecture are the components which encapsulate the game's state. These data structure components are placed at the top since game state is vital for the functioning of the other two groups of components. These ADT components receive no notifications, but respond to requests and emit notifications of internal state changes. ADT notifications are directed to the next level where they are received by both the game logic components and the artists components.

The game logic components request changes of ADT state in accordance with game rules and interpret ADT state

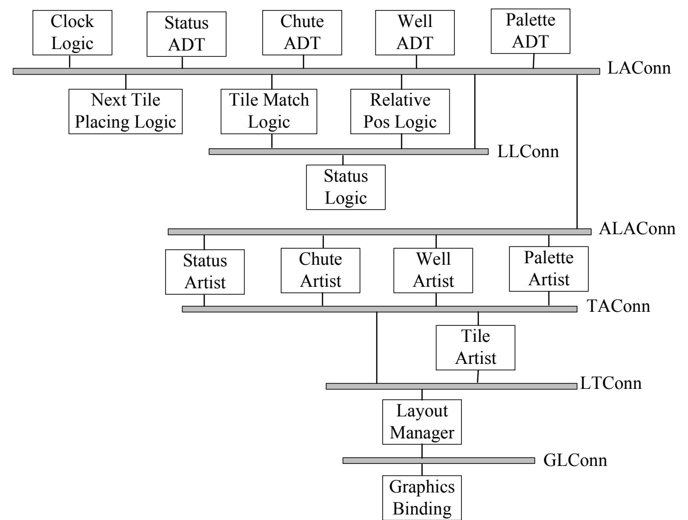

Fig. 1. KLAX Architecture in the C2-Style

change notifications to determine the state of the game in progress. This notification is detected by the status logic causing the number of lives to be decremented.

The artist components also receive notifications of ADT state changes, causing them to update their depictions. Each artist maintains the state of a set of abstract graphical objects which, when modified, send state change notifications in hope that a lower-level graphics component will render them. The tile artist provides a flexible presentation level for tiles. Artists maintain information about the placement of abstract tile objects. The tile artist intercepts any notifications about tile objects and recasts them to notifications about more concrete drawable objects. The layout manager component receives all notifications from the artists and offsets any coordinates to ensure that the game elements are drawn in the correct juxtaposition.

The graphics binding component receives all notifications about the state of the artists' graphical objects and translates them into calls to a window system. User events, such as a key press, are translated into requests to the artist components.

The component dependency graph of the KLAX system is shown as Fig. 2. Where GraphicsBinding and StatusADT etc. are components which are represented circles, LAConn and TAConn etc. are connectors which are represented rectangles, small solid rectangles represent interface of component and connector, thick solid edges represent dependency edge from component (connector) to connector (component) that connect an interface of a component (connector) to an interface of a corresponding connector (component). Thick dotted edges represent dependency edge from connector to connector that it connects an interface of a connector and an interface of a corresponding connector. Thin dotted edges represent additional dependency edge that it connects two interfaces or interface within a component or connector.

B. Component dependency path

Software architecture has many new characteristics, such as component, connector and so on. All of these characteristics have an impact on interactions among component, connector and component dependency path generation.

Fig. 2. G_{CD} of KLAX System

Definition 2 (Component dependency path). Let G_{CD} = (N, E) be a component dependency graph for C2-style architecture, a component dependency path is a sequence nodes $C_1 \to C_2 \to \ldots \to C_m$ such that $(C_i, C_j) \in$ E $\wedge$ $C_1 \in$ Comp $\wedge$ $C_m \in$ Comp for all i, 1≤i≤j≤m, called CDP for short. The number of dependency edges in CDP is the length of CDP.

Note, the CDP has two forms according to the type of dependency edges, one is all of dependency edges from the beginning of top interface of component and connector to the end of bottom interface of component and connector, the other is all of dependency edges from the beginning of bottom interface of component and connector to the end of top interface of component and connector.

For example in Fig. 2, because (LayoutManager, GLConn) $\in$ E, (GLConn, GraphicsBinding) $\in$ E, thus, LayoutManager $\to$ GLConn $\to$ GraphicsBinding is a CDP of length is 2. Similarly, LayoutManager $\to$ GLConn $\to$ TileArtist $\to$ TAConn $\to$ StatusArtist $\to$ ALAConn $\to$ LAConn $\to$ ClockLogic is a CDP of length is 7.

The CDP of length n is similar to the Length_N path coverage defined by Li [11] for conventional software, which requires coverage of all subpaths in the program's flow-graph of length less than or equal to n. However, G_{CD} has many start nodes and terminal nodes, and a fixed length of CDP is not necessarily between any two nodes. As the length of the CDP increases, the number of possible contexts also increases.

V. Component Dependency Path Coverage Criteria

Having created representations for G_{CD} among components and connectors, we are ready to define the component dependency path coverage criteria. We will first define coverage criteria between components, i.e., direct component dependency coverage criterion and indirect component dependency coverage criterion, and then for lengths among components, i.e., Length-N component dependency coverage criterion.

A. Direct component dependency path coverage criterion

Intuitively, direct component dependency path coverage requires each component dependency path between components does not contain other components in G_{CD} to be analyzed at least once. Such a requirement is necessary to check whether each direct component dependency path executes as expected.

Definition 3 (Direct component dependency path coverage criterion). We call a set of component dependency paths CDPs to satisfy the direct component dependency path coverage criterion, denoted as DCDPCC, for the CDP $C_1 \to C_2 \to \ldots \to C_m$ in G_{CD} of the C2-style architecture, if and only if each $(C_i, C_j) \in$ DEConn-Conn for i = 2, 3, ..., m-2, j = i+1, i+2, ..., m-1.

Intuitively, the DCDPCC requires that all CDPs of length 2 will be covered. For example in Fig. 2, for CDP LayoutManager $\to$ GLConn $\to$ GraphicsBinding, its length is 2. According to DCDPCC, this CDP will be covered. Similarly, LayoutManager $\to$ LTConn $\to$ TAConn $\to$ StatusArtist also will be covered.

Let $DCDP'$ represents the set of direct component dependency paths that covered by test suites set TS, $||DCDP'||$ represents the number of elements in $DCDP'$, $||EP(DCDP(G_{CD}))||$ represents the number of direct component dependency paths in G_{CD}, then direct component dependency path coverage rate is calculated as follows:

$$R_{DCDP} = \frac{||DCDP'||}{||EP(DCDP(G_{CD}))||} \times 100\% \qquad (1)$$

B. Indirect component dependency path coverage criterion

Indirect component dependency path coverage criterion requires that after a component dependency path between components has been analyzed, all the indirect component dependency paths between components should be analyzed at least once.

Definition 4 (Indirect component dependency path coverage criterion). We call a set of component dependency paths CDPs to satisfy the indirect component dependency path

coverage criterion, denoted as ICDPCC, for the CDP $C_1 \rightarrow C_2 \rightarrow \ldots \rightarrow C_m$ in G_{CD} of the C2-style architecture, if and only if each (C_i, C_j) ∈ DEConn-Comp ∨ DEConn-Conn ∨ DEComp-Conn for i = 2, 3, . . ., m-2, j = i+1, i+2, . . ., m-1.

Note that the ICDPCC requires that all CDPs of length greater than or equal to 2 will be covered. For example in Fig. 2, for CDP LayoutManager → LLConn → TileArtist → TAConn → StatusArtist → ALAConn → LAConn → ClockLogic, its length is 7. According to ICDPCC, this CDP will be covered.

Let $ICDP'$ represents the set of indirect component dependency paths that covered by test suites set TS, $||ICDP'||$ represents the number of elements in $ICDP'$, $||EP(ICDP(G_{CD}))||$ represents the number of indirect component dependency paths in G_{CD}, indirect component dependency path coverage rate is calculated as follows:

$$R_{ICDP} = \frac{||ICDP'||}{||EP(ICDP(G_{CD}))||} \times 100\% \qquad (2)$$

C. Length-N component dependency path coverage criterion

When there are a number of connections between connector and component in G_{CD}, direct and indirect component dependency path coverage criteria are often impractical. For nodes C_i and C_j, if the number of dependency edges such that (C_i, C_j) ∈ E is L, the number of CDPs is greater than 2^L, when L>10, the test will become more difficult. In order to enhance the feasibility of CDP coverage, we present the Length-N component dependency path coverage criterion.

Definition 5 (Length-N component dependency path coverage criterion). We call a set of component dependency paths CDPs to satisfy the Length-N component dependency path coverage criterion, denoted as L_NCDPCC, for the CDP $C_1 \rightarrow C_2 \rightarrow \ldots \rightarrow C_m$ in G_{CD} of the C2-style architecture, if and only if the length of CDP is less than or equal to N and each (C_i, C_j) ∈ DEConn-Comp ∨ DEConn-Conn ∨ DEComp-Conn for i = 2, 3, . . ., m-2, j = i+1, i+2, . . ., m-1.

Note, the L_NCDPCC requires that all CDPs of length greater than or equal to 2 will be tested. For example in Fig. 2, for CDP LayoutManager → LLConn → TileArtist, its length is 2, LayoutManager → LLConn → TileArtist → TAConn → StatusArtist, its length is 4, and LayoutManager → LLConn → TileArtist → TAConn → StatusArtist → ALAConn → LAConn → ClockLogic, its length is 7. According to L_7CDPCC, these CDPs will be covered.

Let $LCDP'_N$ represents the set of CDPs of length n that covered by test suites set TS, $||LCDP'_N||$ represents the number of elements in $LCDP'_N$, $||EP(LCDP_N(G_{CD}))||$ represents the number of CDPs of length n in G_{CD}, then component dependency path of length n coverage rate is calculated as follows:

$$R_{L_NCDP} = \frac{||LCDP'_N||}{||EP(LCDP_N(G_{CD}))||} \times 100\% \qquad (3)$$

Our component dependency coverage criteria have a clear subsumption relationships when k ≥ 2. The L_NCDPCC subsumes the ICDPCC, while the ICDPCC subsumes the DCDPCC.

VI. Algorithms of Component Dependency Path Coverage Rate

Having formally presented DCDPCC, ICDPCC, and L_NCDPCC, we now present algorithms to evaluate the coverage of a test suite using these CDPCC.

A. Algorithm to determine R_{DCDP}

Algorithm RDCDPA can be used to calculate the direct component dependency path coverage rate. The main idea of direct component dependency path coverage rate algorithm can be briefly stated as follows: Let set DCDP' to save the test suites TS covering the direct component dependency path. The start of the algorithm, the DCDP' is empty.

Algorithm 1 RDCDPA
Require: G_{CD}
Ensure: Direct component dependency path coverage rate
Begin
1 DCDP' = ∅.
2 If the test suite ts in TS have been visited, goto Step 4, otherwise, remove the 1 unvisited test suite ts from TS, and its mark as visit.
3 If the direct component dependency path in ts have visited, goto Step 1, otherwise, according to the order from the ts remove 1 has not visited to the direct component dependency path dcdp, and marks it as has visit.
4 Let DCDP' = DCDP' ∪ dcdp, goto Step 2.
5 Output ||DCDP'||/||DCDP|| × 100%.
End RDCDPA

We employ the G_{CD} shown in Fig. 2 to demonstrate algorithm RDCDPA. Let us consider examples showing the computation of the direct component dependency path coverage rate for component TA.

First, DCDP' = ∅. According to steps 2-3, we get dcdp = {TA → LTConn → LM}, so, DCDP' = {TA → LTConn → LM}.

Repeat the steps 3-4, we get DCDP' = {TA → LTConn → LM, TA → TAConn → SA, TA → TAConn → CA, TA → TAConn → WA, TA → TAConn → PA}. Therefore, the number of direct component dependency paths for component TA is 5.

While the all of number of direct component dependency paths is 104. Hence, by step 4, the R_{DCDP} = 5 / 104 × 100% = 4.81% for component TA.

B. Algorithm to determine R_{ICDP}

Algorithm RICDPA can be used to calculate the indirect component dependency path coverage rate. The main idea of indirect component dependency path coverage rate algorithm can be briefly stated as follows: Let set ICDP' to save the test suites TS covering the indirect component dependency path. The start of the algorithm, the ICDP' is empty.

We again employ the G_{CD} shown in Fig. 2 to demonstrate algorithm RICDPA. Let us consider examples showing the computation of the indirect component dependency path coverage rate for component TA.

First, ICDP' = ∅. According to steps 2-3, we get icdp = {TA → LTConn → LM}, so, ICDP' = {TA → LTConn → LM}.

Algorithm 2 RICDPA

Require: G_{CD}

Ensure: Indirect component dependency path coverage rate

Begin

1 ICDP' = $\emptyset$.

2 If the test suite ts in TS have been visited, goto Step 4, otherwise, remove the 1 unvisited test suite ts from TS, and its mark as visit.

3 If the indirect component dependency path in ts have visited, goto Step 1, otherwise, according to the order from the ts remove 1 has not visited to the indirect component dependency path icdp, and marks it as has visit.

4 Let ICDP' = ICDP' $\cup$ icdp, obtain dependency edge DE_t from icdp in G_{CD}, let ICDP' = ICDP' $\cup$ DE_t, goto Step 2.

5 Output ||ICDP'||/||ICDP|| $\times$ 100%.

End RICDPA

Repeat the steps 3-4, we get ICDP' = {TA $\rightarrow$ LTConn $\rightarrow$ LM, TA $\rightarrow$ LTConn $\rightarrow$ LM $\rightarrow$ GLConn $\rightarrow$ GB, TA $\rightarrow$ TAConn $\rightarrow$ SA, TA $\rightarrow$ TAConn $\rightarrow$ CA, TA $\rightarrow$ TAConn $\rightarrow$ WA, TA $\rightarrow$ TAConn $\rightarrow$ PA, TA $\rightarrow$ TAConn $\rightarrow$ SA $\rightarrow$ ALAConn $\rightarrow$ LAConn $\rightarrow$ CL, . . ., TA $\rightarrow$ TAConn $\rightarrow$ SA $\rightarrow$ ALAConn $\rightarrow$ LAConn $\rightarrow$ PADT, TA $\rightarrow$ TAConn $\rightarrow$ CA $\rightarrow$ ALAConn $\rightarrow$ LAConn $\rightarrow$ CL, . . ., TA $\rightarrow$ TAConn $\rightarrow$ CA $\rightarrow$ ALAConn $\rightarrow$ LAConn $\rightarrow$ PADT, TA $\rightarrow$ TAConn $\rightarrow$ WA $\rightarrow$ ALAConn $\rightarrow$ LAConn $\rightarrow$ CL, . . ., TA $\rightarrow$ TAConn $\rightarrow$ WA $\rightarrow$ ALAConn $\rightarrow$ LAConn $\rightarrow$ PADT, TA $\rightarrow$ TAConn $\rightarrow$ PA $\rightarrow$ ALAConn $\rightarrow$ LAConn $\rightarrow$ CL, . . ., TA $\rightarrow$ TAConn $\rightarrow$ PA $\rightarrow$ ALAConn $\rightarrow$ LAConn $\rightarrow$ PADT}. Therefore, the number of indirect component dependency paths for component TA is 26.

While the all of number of indirect component dependency paths is 350. Hence, by step 4, the R_{ICDP} = 26 / 350 $\times$ 100% = 7.43% for component TA.

C. Algorithm to determine $R_{L_N CDP}$

In order to calculate the component dependency path of length n coverage rate, we must calculate two values:

- $||CDP_N||$: The number of component dependency paths of length n contained test cases set TS.
- ||CDP||: The number of component dependency paths of length n contained in G_{CD}.

So, $||CDP_N||$/||CDP|| $\times$ 100% is the component dependency path of length n coverage rate.

Algorithm 3 RLCDPA

Require: G_{CD}

Ensure: Component dependency path of length n coverage rate

Begin

1 CDP_N = $\emptyset$.

2 If the test suite ts in TS have been visited, goto Step 3, otherwise, remove the 1 unvisited test suite ts from TS, and its mark as visit.

3 Obtain component dependency path of length less than or equal to N, let CDP_N = $CDP_N \cup CDP'_N$, goto Step 1.

4 Output $||CDP_N||$.

End RLCDPA

Algorithm RLCDPA can be used to calculate the component dependency path of length n coverage rate. The main idea of component dependency path of length n coverage rate algorithm can be briefly stated as follows: Let set CDP_N to save the test suites TS covering the component dependency path of length n. The start of the algorithm, the CDP_N is empty.

In order to calculate the ||CDP||, we define recursive process CDPath(C_i, N), the process returns the set of component dependency paths from component node C_i and length equal to N. The main steps of the process are shown as algorithm Compute the $||CDP_N||$.

Algorithm 4 Compute the $||CDP_N||$

Require: G_{CD}

Ensure: The number of component dependency paths of length n contained test cases set TS

Begin

11 Obtain C_i for tail node of the component dependency path P = $\{p_1, p_2, \ldots, p_m\}$, if P = $\emptyset$, returns empty set, otherwise, if N = 2, returns P.

12 if N $>$ 2, defines and initialize the set K is empty, save component dependency path of length equal to N for start node C_i to K.

13 If the component dependency path in P have visited, goto Step 17, otherwise, remove 1 has not visited to the component dependency path p_k (1$\leq$k$\leq$m), and marks it as has visit.

14 Obtain target node C'_i of p_k.

15 Obtain set of component dependency paths K' = CDPath(C'_i, N-1) from start node C'_i and length is equal to N-1.

16 If K' $\neq \emptyset$, add component dependency path p_k before each element in K', let K = K $\cup$ K', goto Step 13, otherwise goto Step 13.

17 Return K.

End

The ||CDP|| can be calculated by recursive process CDPath, the start of algorithm, the CDP is empty. The main steps of the process are shown as algorithm Compute the ||CDP||.

Algorithm 5 Compute the ||CDP||

Require: G_{CD}

Ensure: The number of component dependency paths of length n contained in G_{CD}

Begin

21 CDP = $\emptyset$.

22 If the components in G_{CD} have visited, goto Step 24, otherwise, remove 1 has not visited to the component C_j, and marks it as has visit.

23 Obtain all of the component dependency path from C_j, for each N' $\in$ [1..N], compute CDP' = CDPath(C_j, N'), let CDP = CDP $\cup$ CDP', goto Step 22.

24 Return ||CDP||.

End

We again employ the G_{CD} shown in Fig. 2 to demonstrate algorithm RLCDPA. Let us consider examples showing the

TABLE II
TOTAL NUMBER OF COMPONENT DEPENDENCY PATH IN KLAX SYSTEM

Component name	DCDPCC	ICDPCC	L_NCDPCC							
			N=2	N=3	N=4	N=5	N=6	N=7	N=8	N=9
GraphicsBinding	1	50	1	-	6	-	30	-	-	50
LayoutManager	6	50	6	-	30	-	-	50	-	-
TileArtist	5	26	5	-	6	26	-	-	-	-
StatusArtist	7	10	1	7	8	9	10	-	-	-
ChuteArtist	7	10	1	7	8	9	10	-	-	-
WellArtist	7	10	1	7	8	9	10	-	-	-
PaletteArtist	7	10	1	7	8	9	10	-	-	-
StatusLogic	7	17	2	7	17	-	-	-	-	-
NextTilePlacingLogic	5	5	5	-	-	-	-	-	-	-
TileMatchLogic	6	6	6	-	-	-	-	-	-	-
RelativePosLogic	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

computation of the component dependency path of length 5 coverage rate for component TA.

First, CDP_5 = $\emptyset$. According to step 2-3, we get CDP of length 2, CDP_5 = {TA → LTConn → LM}, so, CDP_5 = {TA → LTConn → LM}. Repeat the step 1-2, we get CDP of length 2, CDP_5 = {TA → LTConn → LM, TA → TAConn → SA, TA → TAConn → CA, TA → TAConn → WA, TA → TAConn → PA}.

Similarly, we get CDP of length 4, CDP_5 = {TA → LTConn → LM → GLConn → GB} and CDP of length 5, CDP_5 = {TA → TAConn → PA, TA → TAConn → SA → ALAConn → LAConn → CL, ..., TA → TAConn → SA → ALAConn → LAConn → PADT, TA → TAConn → CA → ALAConn → LAConn → CL, ..., TA → TAConn → CA → ALAConn → LAConn → PADT, TA → TAConn → WA → ALAConn → LAConn → CL, ..., TA → TAConn → WA → ALAConn → LAConn → PADT, TA → TAConn → PA → ALAConn → LAConn → CL, ..., TA → TAConn → PA → ALAConn → LAConn → PADT}. Therefore, the number of component dependency paths of length 5 for component TA is 26.

The following is a calculation process of CDP of length smaller than or equal to 5.

First, we get set of CDPs of length equal to 5 by step 4, P = {BG → GLConn → LM → LTConn → TAConn → SA, BG → GLConn → LM → LTConn → TAConn → CA, BG → GLConn → LM → LTConn → TAConn → WA, BG → GLConn → LM → LTConn → TAConn → PA}. By step 5, CDP = $\emptyset$. By steps 6-9, recursive call CDPath(C_i', N-1) on BG → GLConn → LM → LTConn → TAConn → SA. We get CDP' = {BG → GLConn → LM}, so CDP = {BG → GLConn → LM}. Continue steps 6-9, Similarly, CDP = {BG → GLConn → LM → LTConn → TAConn → SA, BG → GLConn → LM → LTConn → TAConn → CA, BG → GLConn → LM → LTConn → TAConn → WA, BG → GLConn → LM → LTConn → TAConn → PA, BG → GLConn → LM}. Similarly, we get other CDPs of length smaller than or equal to 5.

While the all of number of component dependency paths of length smaller than or equal to 5 is 132. Hence, the R_{L_5CDP} = 26 / 132 × 100% = 19.70% for component TA.

VII. EXPERIMENTAL RESULTS

In order to test the effectiveness and performance of the proposed component dependency path coverage method, we implemented the dependency-based in C2-style architecture path testing system (DC2PTS) platform, and carried out lots of experiments. We choose KLAX system as the application under test, extract 16 components and the 6 connectors in KLAX system, and determine the total number of component dependency paths. The total number of component dependency paths is shown in Table II. The first column represents the component name of KLAX system, the second column represents the number of direct component dependency path from component of the first column on DCDPCC, the third column represents the number of indirect component dependency path from component of the first column on ICDPCC, and the fourth column represents the number of component dependency path from component of the first column on L_NCDPCC, the symbol "-" means that there isn't exist component dependency path of length n for component of the first column.

From the Table II, we obtain the two conclusions as follows:

- The total number of indirect component dependency paths is greater than direct component dependency paths for component. Note that indirect component dependency paths subsume direct component dependency paths, if all indirect component dependency paths are tested, then so are all direct component dependency paths.
- The total number of component dependency paths of length n grows with increasing length. If all component dependency paths of length n are tested, then so are all component dependency paths of length i (i≤N-1).

Table III gives the component dependency path coverage rate for each of the component in KLAX system. The symbol

TABLE III
COMPONENT DEPENDENCY PATH COVERAGE RATE IN KLAX SYSTEM

Component name	R_{DCDP} (%)	R_{ICDP} (%)	$R_{L_N CDP}$ (%)							
			N=2	N=3	N=4	N=5	N=6	N=7	N=8	N=9
GraphicsBinding	0.96	14.29	2.00	-	4.26	-	18.75	-	-	25.00
LayoutManager	5.77	14.29	12.00	-	21.28	-	-	31.25	-	-
TileArtist	4.81	7.43	10.00	-	4.26	19.70	-	-	-	-
StatusArtist	6.73	2.86	2.00	9.33	5.67	6.82	6.25	-	-	-
ChuteArtist	6.73	2.86	2.00	9.33	5.67	6.82	6.25	-	-	-
WellArtist	6.73	2.86	2.00	9.33	5.67	6.82	6.25	-	-	-
PaletteArtist	6.73	2.86	2.00	9.33	5.67	6.82	6.25	-	-	-
StatusLogic	6.73	4.86	4.00	9.33	12.06	-	-	-	-	-
NextTilePlacingLogic	4.81	1.43	10.00	-	-	-	-	-	-	-
TileMatchLogic	5.77	1.71	12.00	-	-	-	-	-	-	-
RelativePosLogic	5.77	1.71	12.00	-	-	-	-	-	-	-
ClockLogic	7.69	8.57	6.00	10.67	7.09	10.61	11.25	13.75	20.00	15.00
StatusADT	7.69	8.57	6.00	10.67	7.09	10.61	11.25	13.75	20.00	15.00
ChuteADT	7.69	8.57	6.00	10.67	7.09	10.61	11.25	13.75	20.00	15.00
WellADT	7.69	8.57	6.00	10.67	7.09	10.61	11.25	13.75	20.00	15.00
PaletteADT	7.69	8.57	6.00	10.67	7.09	10.61	11.25	13.75	20.00	15.00

"-" means that there isn't exist component dependency path of length n coverage rate for component. For some applications such as component GraphicsBinding, ClockLogic, StatusADT, ChuteADT, WellADT, and PaletteADT, direct and indirect component dependency path coverage rate increases from 39.41% to 57.14%. The component dependency path from length 9 to length 2 coverage rate decreases from 100% to 32%. However, for component StatusArtist, ChuteArtist, WellArtist, and PaletteArtist, the component dependency path of length 2, 3, 4, 5, and 6 coverage rate increases from 8% to 37.32%. The reason is, component at the top/bottom of software architecture due to large number of components interact with other levels, the number of length of component dependency path also increases relatively, its the component dependence path coverage rate is relatively high. But the component at the middle level, its number of component dependency paths less than top/bottom component, that is the number of component dependency paths will be a decrease in the middle level, making the component dependency path coverage rate is relatively low.

VIII. CONCLUSION

Component-based software is an important pattern in software developing. Effective testing can ensure the software quality. This paper presents a component dependency path coverage technology based on C2-style architecture. First, it describes software architecture through C2-style, then represents software architecture through component dependency graph G_{CD}, and abstracted the behavior of interactive between components and connectors. Formalized the component dependency path, generated the component dependency path coverage set that covered the C2-style architecture according to the component dependency path coverage criteria and algorithms. The assessment of results for the tester can help to understand each properties of component dependency path coverage criteria. The difference between using different component dependency path coverage criteria provide reference in practice, at the same time, so that we will do further research on the adequacy criteria.

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] 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.

[3] 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.

[4] 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.

[5] L. J. Lun and X. Chi, "Dependency coverage for C2-style architecture," *Journal of Computational Information System*, vol. 10, no. 7, pp. 3039-3048, 2014.

[6] 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.

[7] J. A. Stafford, A. L. Wolf and M. Caporuscio, "The application of dependence analysis to software architecture descriptions," *Lecture Notes in Computer Science*, vol. 2804, pp. 52-52, 2003.

[8] M. Marré and A. Bertolino, "Using spanning sets for coverage testing," *IEEE Trans. Softw. Engi.*, vol. 29, no. 11, pp. 974-984, 2003.

[9] J. Costa and J. Monteiro, "Observability-based coverage-directed path search using PBO for automatic test vector generation," in *Proceedings of the IFIP/IEEE International Conference on Very Large Scale Integration*, October 2009, pp. 153-158.

[10] Y. Z. Gong, W. Zhang and Q. L. Lu, "Making software testing system oriented faults," in *Proceedings of the 6th International Conference on Computer Aided Industial Design & Conceptual Design*, May 2005, pp. 840-844.

[11] 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.

[12] L. Brim, I. Černá, P. Vařeková and B. Zimmerova, "Component-interaction automata as a verification-oriented component-based system specification," in *Proceedings of the 2005 Conference on Specification and Verification of Component-based Systems*, September 2005, pp. 31-38.

[13] Y. Wu, D. Pan and M. H. Chen, "Techniques for testing component-based software," in *Proceedings of the Seventh IEEE International Conference on Engineering of Complex Computer Systems*, June 2001, pp. 222-232.

[14] B. X. Li, "Managing dependencies in component-based systems based on matrix model," in *Proceedings of Net.Object.Days*, September 2003, pp. 22-25.
[15] I. -C Yoon, A. Sussman, A. M. Memon and A. Porter, "Direct-dependency-based software compatibility testing," in *Proceedings of the 22nd IEEE International Conference on Automated Software Engineering*, November 2007, pp. 409-412.
[16] 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.
[17] J. Gao, R. Espinoza and J. He, "Testing coverage analysis for software component validation," in *Proceedings of Annual International Computer Software and Applications Conference*, July 2005, pp. 463-470.
[18] 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.
[19] 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.
[20] H. Muccini, M. D. Dias and D. J. Richardson, "Towards software architecture-based regression testing," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1-7, 2005.
[21] L. J. Lun, X. Chi and X. M. Ding, "Edge coverage analysis for software architecture testing," *Journal of Software*, vol. 7, no. 5, pp. 1121-1128, 2012.
[22] N. Medvidovic, D. S. Rosenblum, D. F. Redmiles and J. E. Robbins, "Modeling software architectures in the unified modeling language," *ACM Transactions on Software Engineering and Methodology*, vol. 11, no. 1, pp. 2-57, 2002.
[23] X. F. Zhang, J. Fiskio-lasseter and M. Young, "Flow analyses for abstraction of architectural structure and behavior," in *Technical Report CIS-TR-03-01*, 2003.