Abstract—Software architecture has already become one of the primary research areas in software engineering recently and how to test software architecture automatically, effectively and adequately is a difficulty in issues about software architecture. Currently, many people are doing the research of software architecture analyze, evaluation, testing and verification techniques, and some representative testing strategies are proposed to test software architecture. But, traditional software testing methods can not be used directly to solve the test issues of software architecture, either some techniques are needed to improve the traditional methods or new software architecture testing techniques are developed to solve the test issues related to software architecture. Dependency analysis is an important method to test, analyze, understand, and maintain programs. A new kind of dependency analysis method for C2-style architecture is developed. A set of dependency relationships is defined corresponding to the relationships among C2-style architecture elements. The C2-style element dependency graph (C2-EDG) of C2-style architecture can be constructed from these dependency relationships. Based on the C2-EDG, both architecture dependency coverage testing and metrics are further given as its two applications, and discusses the equivalence of existing methods.

Index Terms—software architecture testing; software metrics; C2-style; dependency analysis; coverage criteria

I. INTRODUCTION

Software architecture is the highest abstract description of a software design, which is defined at the initial stages of the software development. Software architectures are commonly described in terms of three basic abstractions: components, connectors, and configurations. Components represent a wide range of different elements, from a single client to a database, and have an interface (made up of ports) used to communicate the component with the external environment. Connectors represent communication elements between components. Configuration describes how components and connectors are wired.

The complexity of software architecture embodies dependency relationships between component and connector, architecture dependency describes the dependency relationships between component and connector that are implicitly determined by the control and data flows in the software architecture. Architecture dependency analysis [1,2] is a technique to identify and determine various dependency relationships in the architecture specification and to represent them in some explicit forms convenient for many applications. So a component or connector change will affect the other component or connector. It also makes testing and metrics more complex architecture. The dependency analysis method is used to help in reducing the number of experiments necessary to cover the architecture interface.

In this paper, a new method to analyze dependencies for C2-style architecture is proposed. 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 relationships among C2-style architecture elements. The C2-style element dependency graph (C2-EDG) of C2-style architecture can be constructed from these dependency relationships. Based on the C2-EDG, both architecture dependency coverage testing and metrics are further given as its two applications, and discusses the equivalence of existing methods.

II. C2-STYLE ARCHITECTURE

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 [3]; at the same time, the rules of the C2-style are broad enough to render it widely applicable.
A C2-style architecture consists of components, connectors, and their constraints. Each component has two connection points, a "top" and a "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 components above and notifications of completed services to components below.

**Definition 2.1** A C2-style architecture can be defined as $C2 = \{\text{Comp}, \text{Conn}\}$, where:
- $\text{Comp} = \{\text{Comp}_1, \text{Comp}_2, \ldots, \text{Comp}_n\}$ is a finite set of components, where $\text{Comp}_i = \{\text{Comp}_i.\text{Ip}_i, \text{Comp}_i.\text{Ip}_i.\text{In}_i, \text{Comp}_i.\text{Ip}_i.\text{In}_o, \text{Comp}_i.\text{Ip}_i.\text{Int}_i, \text{Comp}_i.\text{Ip}_i.\text{Int}_o\}$.
- $\text{Conn} = \{\text{Conn}_1, \text{Conn}_2, \ldots, \text{Conn}_n\}$ is a finite set of connectors, where $\text{Conn}_j = \{\text{Conn}_j.\text{Ip}_j.\text{Ipb}_j, \text{Conn}_j.\text{Ip}_j.\text{Ipt}_j, \text{Conn}_j.\text{Ip}_j.\text{Ipb}_j.\text{In}_j, \text{Conn}_j.\text{Ip}_j.\text{Ipt}_j.\text{In}_j, \text{Conn}_j.\text{Ip}_j.\text{Ipt}_j.\text{Int}_j, \text{Conn}_j.\text{Ip}_j.\text{Ipt}_j.\text{Inb}_j\}$.
- $\text{Ip}_j.\text{Ipb}_j$ or $\text{Ip}_j.\text{Ipt}_j$ is the set of requests received at the bottom side of component or connector. $\text{Ip}_j.\text{Ipb}_j$ or $\text{Ip}_j.\text{Ipt}_j$ is the set of notifications that component or connector emits from its bottom side.
- $\text{Ip}_j.\text{Ipt}_j$ or $\text{Ip}_j.\text{In}_j$ is the set of notifications received on the top side of component or connector. $\text{Ip}_j.\text{Ipt}_j$ or $\text{Ip}_j.\text{In}_j$ is the set of requests sent from its top side.

Fig. 1 represents the external view of a component Comp. Comp. Ipb_i and Comp. Ipt_o are defined by the component’s dialog. They are the requests it will be submitting and notifications it will be handling. Comp. Ipb_o is the notifications the component will be making, reflecting changes to its internal object. Comp. Ip_i is the requests the component accepts.

**Property 1** $\forall (I_p. I_a) DEP_{I_p. I_a} (I_i. I_j) \Rightarrow DEP_{I_p. I_a. I_i. I_j}$

According to the definition 2.2, 2.3, and 2.4, we have:

**Definition 2.4** Let Comp_1 is a component in C2-style architecture, I_p is the top of the Comp_1, I_a is a connector in C2-style architecture, I_a is the bottom of the Comp_2. If there are $\text{DEP}_{I_p. I_a. I_i. I_j}$ and $\text{DEP}_{I_a. I_i. I_j}$, $\text{DEP}_{I_p. I_a. I_i}$, and $\text{DEP}_{I_p. I_a. I_j}$, then $I_p$ and $I_a$ depend on each other, denoted by $\text{DEP}_{I_p. I_a. I_i. I_j}$.

**III. DEPENDENCY RELATIONSHIPS IN THE C2-STYLE**

Dependency relationships at 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.

**A. Dependency Relationship between Interface**

**Definition 2.2** Let Comp_i is a component in C2-style architecture, I_p is the top of the Comp_i, I_a is a connector in C2-style architecture, I_a is the bottom of the Comp_i. If the change of Comp_i. I_p.o affects Comp_i. I_b.i, then Comp_i. I_b.i depends on Comp_i. I_p.o, denoted by $\text{DEP}_{I_p. I_a. I_i. I_j}$.

**Definition 2.3** Let Comp_i is a component in C2-style architecture, I_p is the top of the Comp_i, I_a is a connector in C2-style architecture, I_a is the bottom of the Comp_i. If there are $\text{DEP}_{I_p. I_a. I_i. I_j}$ and $\text{DEP}_{I_a. I_i. I_j}$, $\text{DEP}_{I_p. I_a. I_i}$, and $\text{DEP}_{I_p. I_a. I_j}$, then $I_p$ and $I_a$ depend on each other, denoted by $\text{DEP}_{I_p. I_a. I_i. I_j}$.

**Definition 2.4** Let Comp_1 is a component in C2-style architecture, I_p is the top of the Comp_1, I_a is the interface of the Comp_1, I_b is the interface of the Comp_2. If there are $\text{DEP}_{I_p. I_a. I_i. I_j}$ and $\text{DEP}_{I_a. I_i. I_j}$, $\text{DEP}_{I_p. I_a. I_i}$, and $\text{DEP}_{I_p. I_a. I_j}$, then $I_p$ and $I_a$ depend on each other, denoted by $\text{DEP}_{I_p. I_a. I_i. I_j}$.
If there is DEP np(Comp1, Ipb_i) or DEP np(Conn2, Int_i), then Comp1 depends on Conn2, denoted by DEP np(Comp1, Conn2).

Definition 2.8 Let Comp1 is a component in C2-style architecture, Conn1 is a connector in C2-style architecture. If there is DEP np(Comp1, Ipb_i) or DEP np(Conn1, Int_i), then Conn1 depends on Comp1, denoted by DEP np(Conn1, Comp1).

Definition 2.9 Let Comp1 is a component in C2-style architecture, Ipb is the interface of the Comp1, Conn1 is a connector in C2-style architecture, Int is the interface of the Conn1. If there is DEP np(Comp1, Ipb_i) or DEP np(Conn1, Int_i), then Comp1 depends on Conn1, denoted by DEP np(Comp1, Conn1).

According to the definition 2.7, 2.8, and 2.9, we have:

Property 3 \( \forall (Comp, Conn, \Delta) \), DEP np(Comp, Conn) if and only if DEP np(Conn, Comp).

C. Dependency Relationship between Connector

Definition 2.10 Let Conn1 and Conn2 are two connectors in C2-style architecture, Ipb is the interface of the Conn1, Int is the interface of the Conn2. If there are DEP mn(Conn1, Ipb_o, Conn2, Ipb_i) and DEP mn(Conn2, Ipb_o, Conn1, Ipb_i), then Conn2 depends on Conn1, denoted by DEP mn(Conn2, Conn1).

Definition 2.11 Let Conn1 and Conn2 are two connectors in C2-style architecture. If there are DEP mn(Conn1, Conn2) and DEP mn(Conn2, Conn1), then Conn1 and Conn2 depend on each other, denoted by DEP mn(Conn1, Conn2).

According to the definition 2.10 and 2.11, we have:

Property 4 \( \forall (Conn, Conn) \), DEP mn(Conn) if and only if DEP mn(Conn, Conn).

D. Dependency Relationship in Component and Connector

Definition 2.12 Let Comp1 is a component in C2-style architecture. If there is a bottom of Comp1, or a top of Comp1 depends on a bottom of Comp2, denoted by DEP d(Comp1, Comp2).

Definition 2.13 Let Conn1 is a connector in C2-style architecture. If there is a bottom of Conn1 depends on a top of Conn2, or a top of Conn1 depends on a bottom of Conn2, denoted by DEP d(Conn1, Conn2).

IV. C2-STYLE ELEMENT DEPENDENCY GRAPH

The C2-style element dependency graph is a digraph whose node represents component or connector, and edge represents possible information flows between component and connector in the C2-ADL architecture specification.

Definition 2.14 Let C2 = (Comp, Conn) is a C2-style architecture, the C2-style element dependency graph for the C2-style architecture denoted by C2-EDG = (V, E), where:

V = \(<Comp, Conn, Ipt, Ipb, Int, Inb>\). Comp represents the set of components in C2-style. Conn represents the set of connectors in C2-style.

Ipt represents the set of top interfaces of component in C2-style. Comp, Ipb_o represents the set of requests sent from its top side of a component Comp, Comp, Ipb_i represents the set of notifications received on the top side of a component Comp.

Ipb represents the set of bottom interfaces of component in C2-style. Comp, Inb_o represents the set of requests sent from its bottom side of a component Comp, Comp, Inb_i represents the set of notifications received on the bottom side of a component Comp.

Ir represents the set of top interfaces of connector in C2-style. Conn, Ipb_o represents the set of requests sent from its top side of a connector Conn, Conn, Ipb_i represents the set of notifications received on the top side of a connector Conn.

Ipb represents the set of bottom interfaces of connector in C2-style. Conn, Ipb_o represents the set of requests sent from its bottom side of a connector Conn, Conn, Ipb_i represents the set of notifications received on the bottom side of a connector Conn.

E = \(<Comp, Ipt_o, Conn, Ipb_i> \lor <Conn, Ipb_o, Comp, Ipt_i> \lor <Comp, Ipb_o, Conn, Ipt_i> \lor <Conn, Inb_o, Comp, Inb_i> \lor <Conn, Ipb_i, Conn, Ipb_o> \lor <Conn, Ipb_o, Conn, Inb_i> \lor <Conn, Ipb_i, Conn, Inb_o> \lor <Conn, Ipb_o, Conn, Ipb_i> \lor <Conn, Inb_i, Conn, Inb_o> \lor <Conn, Ipb_i, Conn, Ipb_o> \lor <Conn, Ipb_o, Conn, Ipb_i> \lor <Conn, Inb_i, Conn, Inb_o> \lor <Conn, Ipb_i, Conn, Ipb_o> \lor <Conn, Ipb_o, Conn, Ipb_i> \lor <Conn, Ipb_i, Conn, Ipb_o> \lor <Conn, Inb_i, Conn, Inb_o> \lor <Conn, Ipb_i, Conn, Ipb_o>

\( \lor <Conn, Ipb_o, Conn, Ipb_i> \lor <Conn, Ipb_i, Conn, Ipb_o> \lor <Conn, Ipb_o, Conn, Ipb_i> \lor <Conn, Ipb_i, Conn, Ipb_o> \lor <Conn, Inb_i, Conn, Inb_o> \lor <Conn, Ipb_i, Conn, Ipb_o> \lor <Conn, Ipb_o, Conn, Ipb_i> \lor <Conn, Ipb_i, Conn, Ipb_o> \lor <Conn, Inb_i, Conn, Inb_o> \lor <Conn, Ipb_i, Conn, Ipb_o> \lor <Conn, Ipb_o, Conn, Ipb_i> \lor <Conn, Ipb_i, Conn, Ipb_o> \lor <Conn, Inb_i, Conn, Inb_o> \lor <Conn, Ipb_i, Conn, Ipb_o> \lor <Conn, Ipb_o, Conn, Ipb_i> \lor <Conn, Ipb_i, Conn, Ipb_o> \lor <Conn, Inb_i, Conn, Inb_o> \lor <Conn, Ipb_i, Conn, Ipb_o>>

© 2013 ACADEMY PUBLISHER


Fig. 3 shows KLAX system [4] architecture representation. It contains sixteen components which are connected by six connectors.

![KLAX Architecture Diagram](image)

**Fig. 3. KLAX architecture in the C2-style**

Fig. 4 shows the C2-EDG of the Fig. 3. Where component expressed with large rectangle, connector expressed with circular bead rectangle, and the interface of component or connector expressed with small solid rectangle. Thick solid edge represents dependency edge from component to connector that connected an interface of a corresponding connector. Thick dashed edge represents dependency edge from connector to component that connected an interface of a corresponding connector. Thick dotted edges represent dependency edges from connectors to connectors that connect an interface of a connector and an interface of a corresponding component. Thick dashed edges represent dependency edges from connectors to connectors that connect an interface of a connector and an interface of a corresponding connector. Thin dotted edges represent additional dependency edges that connect two interface or interface within a component or connector.

In the Fig. 4, there are three types of dependency edge, which are (LayoutManager.Ipb_o, LTConn.Inb_i) represents the dependency edge from component LayoutManager to connector LTConn, (ALAConn.Inb_o, PaletteADT.Ipb_i) represents the dependency edge from connector ALAConn to component PaletteADT, and (LTConn.Inb_o, TAConn.Inb_i) represents the dependency edge from connector LTConn to connector TAConn.

For example, the C2-EDG depicted in the Fig. 4 has:

- **Comp = {GraphicsBinding, LayoutManager, TileArtist, StatusArtist, ...}**
- **Conn = {GLConn, LTConn, TAConn, ALAConn, LAConn, LLConn}**
- **Ipb = {LayoutManager.Ipb_o, LayoutManager.Ipb_i, TileArtist.Ipb_i, StatusArtist.Ipb_o, StatusArtist.Ipb_i, ...}**
- **Inb = {GLConn.Inb_o, GLConn.Inb_i, LTConn.Inb_o, LTConn.Inb_i, ALAConn.Inb_o, ALAConn.Inb_i, LAConn.Inb_o, LAConn.Inb_i, LLConn.Inb_o, LLConn.Inb_i}**
- **E = {<GraphicsBinding.Ipb_o, GLConn.Inb_i>, <GraphicsBinding.Ipb_i, GraphicsBinding.Ipb_o>, ...}**

**V. APPLICATIONS**

Dependency analysis has been widely used in software engineering activities such as software testing [5,6,7,8,9,10], software metrics[11], software maintenance [12], reverse engineering, reengineering, and software reuse. Dependencies among C2-style architecture also can be applied to C2-style coverage testing [13,14].

**A. Dependency Edge Coverage Testing in the C2-Style**

Software architecture with the traditional testing different but linked. The purpose of the test software architecture design is to identify system errors and defects, resulting in guiding the test plan and test code, test cases, which are very different from traditional testing; and the test plan and test cases of software architecture will pass layer of code testing to refine and
Software architecture testing technique includes two aspects, one is software architecture analysis, the other is software architecture testing. Software architecture testing have two main types, first test the software...
architecture, using simulation tools, test software architecture of the interface behavior, or interaction between components, or the communication relationship between the components, analyze the behavior the difference between the target system, the second is based on software architecture object code generated for testing guidance. Software architecture of these two types of coverage testing generation is involved in this core technology.

Testing coverage criteria can be used in one of two ways, as a mechanism to help testers mechanically or manually test generation, or to measure the quality of coverage analysis. Let \( e_{\text{Comp},\text{Conn}} \) represents the dependency edge of C2-EDG, where \( V(i) \) and \( V(j) \) are nodes of C2-EDG, and \( TS_{e_{\text{ Comp},\text{ Conn}}} \) represents a set of test cases created to satisfy \( e_{\text{Comp},\text{Conn}} \).

1. **Dependency edge coverage criteria for component to connector (DEComp-ConnCC)**

   The dependency edge coverage criteria for component to connector requires that every DEP_{pn}(Comp_{1}, Conn_{2}) in C2-EDG be covered by at least one test case.

   **Definition 3.1** For every dependency edge \( e_{\text{Comp},\text{Conn}} \) in C2-EDG, there is at least one test case \( I_{\text{Comp},\text{Conn}} \) such that there is a DEP_{pn} (Comp_{1}, Conn_{2}) induced by \( e_{\text{Comp},\text{Conn}} \), that is a sub-path of the execution trace of C2-EDG.

   The result of dependency edge coverage for component to connector by DEComp-ConnCC can be formalized as follows:
   - \#\# \text{ < Conn}_{1}.\text{Inb}_{o}, \text{Conn}_{2}.\text{Ipt}_{i} > \#\# or
   - \#\# \text{ < Conn}_{1}.\text{Inb}_{o}, \text{Conn}_{2}.\text{Ipt}_{i} > \#\#

2. **Dependency edge coverage criteria for connector to component (DEConn-CompCC)**

   The dependency edge coverage criteria for connector to component requires that every DEP_{np}(Conn_{1}, Conn_{2}) in C2-EDG be covered by at least one test case.

   **Definition 3.2** For every dependency edge \( e_{\text{Conn},\text{Comp}} \) in C2-EDG, there is at least one test case \( I_{\text{Conn},\text{Comp}} \) such that there is a DEP_{np} (Conn_{1}, Conn_{2}) induced by \( e_{\text{Conn},\text{Comp}} \), that is a sub-path of the execution trace of C2-EDG.

   The result of dependency edge coverage for connector to component by DEConn-CompCC can be formalized as follows:
   - \#\# \text{ < Conn}_{1}.\text{Ipt}_{o}, \text{Conn}_{2}.\text{Inb}_{i} > \#\# or
   - \#\# \text{ < Conn}_{1}.\text{Ipt}_{o}, \text{Conn}_{2}.\text{Inb}_{i} > \#\#.

3. **Dependency edge coverage criteria for connector to connector (DEConn-ConnCC)**

   The dependency edge coverage criteria for connector to connector requires that every DEP_{nd}(Conn_{1}, Conn_{2}) in C2-EDG be covered by at least one test case.

   **Definition 3.3** For every dependency edge \( e_{\text{Conn},\text{Conn}} \) in C2-EDG, there is at least one test case \( I_{\text{Conn},\text{Conn}} \) such that there is a DEP_{nd} (Conn_{1}, Conn_{2}) induced by \( e_{\text{Conn},\text{Conn}} \), that is a sub-path of the execution trace of C2-EDG.

   The result of dependency edge coverage for connector to connector by DEConn_ConnCC can be formalized as follows:
   - \#\# \text{ < Conn}_{1}.\text{Ipt}_{o}, \text{Conn}_{2}.\text{Inb}_{i} > \#\# or
   - \#\# \text{ < Conn}_{1}.\text{Ipt}_{o}, \text{Conn}_{2}.\text{Inb}_{i} > \#\#

To verify the C2-style, we carry out experiment [14] on the KLAX system. Tab. I shows the number of dependency edges for three dependency edge coverage criteria. It can be discovered that coverage criteria DEComp-ConnCC covers 24 edges from component to connector according to KLAX system specification. Similar, coverage criteria DEConn-CompCC covers 24 edges from connector to component.

**TABLE I. NUMBER OF DEPENDENCY EDGES COVERAGE IN KLAX**

<table>
<thead>
<tr>
<th>Name of Coverage Criteria</th>
<th>Number of Dependency Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEComp_ConnCC</td>
<td>24</td>
</tr>
<tr>
<td>DEConn_ConnCC</td>
<td>24</td>
</tr>
<tr>
<td>DEConn_ConnCC</td>
<td>6</td>
</tr>
</tbody>
</table>

The following theorem about the number of dependency edges relationship between coverage criteria DEComp-ConnCC and DEConn-CompCC for C2-style architecture.

**Theorem 1** For any C2-style architecture and any set TS of test cases, the number of dependency edges for coverage criteria DEComp-ConnCC is equal to the number of dependency edges for coverage criteria DEConn-CompCC.

**Proof:** If TS satisfies coverage criteria DEComp-ConnCC, then each edge in C2-EDG of C2-style architecture is include in the coverage criteria DEComp-ConnCC, while the same set of test cases TS satisfies coverage criteria DEConn-CompCC, then each edge in C2-EDG of C2-style architecture is include in the coverage criteria DEConn-CompCC.

Thus, this concludes the proof.

**TABLE II. DEPENDENCY COVERAGE RESULT FOR EXPERIMENT**

<table>
<thead>
<tr>
<th>Connector Name</th>
<th>GL Conn</th>
<th>LT Conn</th>
<th>TA Conn</th>
<th>ALA Conn</th>
<th>LL Conn</th>
<th>LA Conn</th>
</tr>
</thead>
<tbody>
<tr>
<td>GLConn</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>LT Conn</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>TA Conn</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>ALAConn</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>LL Conn</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>LAConn</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Tab. II gives the connection relationship of between connector for KLAX system. Where symbol “Yes” satisfy the DEConn_ConnCC relationship, “No” does not satisfy the DEConn_ConnCC relationship.
B. Dependency Edge Coverage Metrics in the C2-Style

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

Let \( ||\text{Comp}|| \) is number of components of C2-style architecture, \( ||\text{Conn}|| \) is number of connectors of C2-style architecture, \( ||e_{\text{Comp},\text{Conn}}|| \) is the number of dependency edges from component to connector, \( ||e_{\text{Conn},\text{Comp}}|| \) is the number of dependency edges from connector to component, \( ||e_{\text{Comp},\text{Conn}}|| \) is the number of dependency edges from connector to connector.

**Definition 3.4** The dependency coverage of component to connector is the total of dependency edge from component to connector divided by the number of components and connectors in C2-style architecture. It is defined as follows:

\[
DEC_{\text{Comp}} = \frac{||\text{Comp}|| \sum_{j=1}^{||\text{Conn}||} ||e_{\text{Comp},\text{Conn}}||}{||\text{Comp}|| + 2 ||\text{Conn}||} \times 100\% \quad (1)
\]

**Definition 3.5** The dependency coverage of connector to component is the total of dependency edge from connector to component divided by the number of components and connectors in C2-style architecture. It is defined as follows:

\[
DEC_{\text{Conn}} = \frac{||\text{Conn}|| \sum_{j=1}^{||\text{Comp}||} ||e_{\text{Conn},\text{Comp}}||}{||\text{Comp}|| + 2 ||\text{Conn}||} \times 100\% \quad (2)
\]

**Definition 3.6** The dependency coverage of connector to connector is the total of dependency edge from connector to connector divided by the number of components and connectors in C2-style architecture. It is defined as follows:

\[
DEC_{\text{Conn}} = \frac{||\text{Conn}|| \sum_{j=1}^{||\text{Conn}||} ||e_{\text{Conn},\text{Conn}}||}{||\text{Comp}|| + 2 ||\text{Conn}||} \times 100\% \quad (3)
\]

**Definition 3.7** The dependency coverage of C2-style architecture is the average of the coverage of component to connector, the coverage of connector to component, and the coverage of connector to connector. It is defined as follows:

\[
DEC_{\text{C2}} = \frac{1}{3} \left( DEC_{\text{Comp}} + DEC_{\text{Conn}} + DEC_{\text{Conn}} \right) \quad (4)
\]

Table III illustrates the computation of three dependency edges test coverage using the Fig. 4.

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

\[ DEC_{\text{C2}} = \frac{1}{3} \left( 85.7\% + 85.7\% + 21.4\% \right) = 64.3\% \]

VI. COMPARISON WITH THE EXISTING METHODS

In this section, we discuss the equivalence of our methods and the existing software architecture testing methods, as well as the conversion method between them.

A. Our Methods are Equivalent to Zhenyi’ Method

Zhenyi and Offutt defined six architecture relations \([9]\) among architecture units: Component(Connector)_Internal_Transfer_Relation(N.interf1, N.interf2), Component(Connector)_Internal_Sequencing_Relation(N.interf1, N.interf2), Component(Connector)_Internal_Relation(N.interf1, N.interf2), N_C_Relation(N.interf1, C.interf1) or C_N_Relation(C.interf1, N.interf1), Direct_Component_Relation(N.interf1, C.interf1, C.interf2, N2.interf2), and Indirect_Component_Relation(N.interf1, C.interf1, C1.interf1, C1.interf2, N2.interf2, N3.interf1). The relations are used to define architecture testing paths, which are then used to define architecture level testing criteria. Through the above analysis, we can see that our proposed technique is equivalent to some coverage methods \([9]\) by Zhenyi and Offutt. Assume Comp, is component, Conn, is connector, and interf is interface. Where:

- \( \text{DEP}_{\text{p}}(\text{Comp}.i_o, \text{Conn}.o_i) \text{ or } \text{DEP}_{\text{pp}}(\text{Comp}.i_o, \text{Conn}.o_i) \) is equivalent to \( \text{Comp}.i_o \text{Conn}.o_i \) Relation(Comp.interf1, Conn.interf1).
- \( \text{DEP}_{\text{p}}(\text{Comp}.i_i, \text{Conn}.o_i) \text{ or } \text{DEP}_{\text{pp}}(\text{Comp}.i_i, \text{Conn}.o_i) \) is equivalent to \( \text{Comp}.i_i \text{Conn}.o_i \) Relation(Conn.interf1, Comp.interf1).
- \( \text{DEP}_{\text{p}}(\text{Comp}.i_i, \text{Conn}.i_o) \text{ and } \text{DEP}_{\text{p}}(\text{Comp}.i_i, \text{Conn}.i_o) \) is equivalent to \( \text{Direct Component Relation} \) (Comp.interf1, Conn.interf1) and Comp_CONN_Relation(Comm.interf1, Comp.interf1).

B. Our Methods are Equivalent to Gao’ Method

Gao et al. proposed an adequate test model \([15]\), known as a CFAQs and D-CFAQs, and presented possible component API-based function operation sequences. And three types of component API-based test coverage criteria can be defined for a given component and its test models. They are: (1) node coverage criteria for each accessible function in a component API interface, (2) link coverage criteria for each link between two nodes, and (3) path coverage criteria for component API-based access sequences between any two nodes. Through the analysis above, we can see that our proposed technique is...
equivalent to some test coverage methods. Let Comp_i is component and Conn_i is connector. Where:

- Node coverage criterion and all-node-coverage criterion and is equivalent to <Comp_i.Inb_o, Conn_i.Inb_i> or <Comp_i.Ipt_o, Conn_i.Int_i> or <Conn_i.Inb_o, Comp_i.Ipt_i> or <Conn_i.Int_o, Comp_i.Ipb_i>.

- Link coverage criterion and all-link coverage criterion is equivalent to the combination of <Comp_1.Ipt_o, Comp_2.Inb_i>, <Comp_2.Int_o, Conn_1.Inb_i>, and <Conn_1.Inb_o, Comp_3.Ipt_i> or <Comp_1.Ipb_o, Comp_2.Inb_i>, <Comp_2.Int_o, Conn_1.Int_i>, and <Conn_1.Inb_o, Comp_3.Ipt_i>.

- If there are relations that connect Comp_1, Comp_2, Comp_3, Conn_1, and Conn_2 together, then the result path Comp_1 − Conn_1 − Comp_2 − Comp_3 of path coverage criterion is equivalent to a number of combinations of <Comp_1.Inb_o, Conn_1.Inb_i>, <Comp_2.Ipt_o, Conn_1.Int_i>, and <Comp_2.Ipb_o, Conn_1.Int_i> or <Comp_1.Ipb_o, Conn_1.Inb_i>, <Comp_2.Ipt_o, Conn_1.Int_i>, and <Conn_2.Inb_o, Conn_1.Int_i>.

- Minimum-set path coverage criterion is equivalent to the minimum of the length of path obtained from component Comp_1 to Comp_2 (Comp_1 ≠ Comp_2) of the C2-EDG, that is min(len(Path_k)), where Path_k is the kth PATH from Comp_1 to Comp_2, len(Path_k) is length of Path_k, len(Path_k) = ∑_{c=1}^C Path_c.

Through discussion above, it can be found that our methods are the most simple, the effectiveness of the method for C2-style software architecture testing and metrics is verified by an application.

VII. RELATED WORK

Traditional dependence analysis has been primarily studied in the context of conventional programming languages. In this languages, it is typically performed using program dependence graphs [16,17]. Traditional dependence analysis though originally proposed for compiler optimization, has also many applications in software engineering activities such as program slicing, testing, debugging, understanding, maintenance and complexity metrics [18,19].

Stafford et al. introduced a software architecture dependence analysis technique [20,21,22], called chaining, to support software architecture development such as debugging and testing. In chaining, links represent the dependence relationships that exist in an architectural specification. Links connect elements of the specification that are directly related, producing a chain of dependencies that can be followed during analysis.

Zhao introduced a new dependence analysis technique [23], named architectural dependence analysis to support software architecture development. In contrast to traditional dependence analysis, architectural dependence analysis is designed to operate on an architectural description of a software system, rather than the source code of a conventional program.

Gao et al. focuses on component test coverage issues, and proposed test models (CFAGs and D-CFAGs) [15] 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 method is provided.

Hashim et al. presented Connector-based Integration Testing for Component-based Systems (CITECB) with an architectural test coverage criteria [24], and describes the test models used that are based on probabilistic deterministic finite automata which are used to represent gate usage profiles at run-time and test execution. It also provides a measuring mechanism of how well the existing test suite are covering the component interactions and provides a test suite coverage monitoring mechanism to reveal the test elements that are not yet covered by the test suites. The model extraction technique used to generate the CITECB test models is a simple and less time consuming process. In addition to that, these test models are able to closely represent the component interactions as they are extracted directly from the system.

Lun et al. presented an edge coverage method [25] for software architecture. They described three type of edge, named component to connector, connector to component, and connector to connector. They use four metrics standard to evaluate the effectiveness of edge coverage criteria.

VIII CONCLUSIONS AND FUTURE WORK

The methods given in this paper shows that dependencies can be grouped based on the identification of components and connectors applicable to all dependencies. From that set of dependencies, a dependency type hierarchy can be produced that will cover all dependencies found in the C2-style architecture. Our initial research indicates that this method provide a more general and unified method to dependency analysis. We have also shown that C2-style element dependency graph provides a powerful method to represent, characterize, and analyze dependencies between the entities in a model. Using C2-EDG, we can establish an abstract model to describe the characteristics of dynamic architecture, it covered all the testing component nodes and reduced scale of testing coverage set, so that test the architecture effectively. We also use dependency edge coverage metrics to evaluate the effectiveness of dependency edge coverage criteria. Therefore, the methods can help successfully for assurance of software quality.

Although our methods can only handle C2-style architecture specifications, we are also considering the use of this method to handle other ADL. We also plan to perform some experiments to show the effectiveness of our methods to support software architecture evolution.
ACKNOWLEDGMENT
Part of this work is supported by the Natural Science Foundation of Heilongjiang Province of China under Grant No. F201036, the Scientific Research Foundation of Heilongjiang Provincial Education Department of China under Grant No. 11551127.

REFERENCES

Lijun Lun was born in Harbin, Heilongjiang Province, China, in 1963. He received his B.S. degree and Master degree in Computer Science and Technology from Harbin Institute Technology of Computer Science and Technology, China, in 1986 and 2000 respectively. Currently, he is a professor, and teaches and conducts research in the areas of software architecture, software testing, and software metrics, etc.
Xin Chi was born in Harbin, Heilongjiang Province, China, in 1990. She is a three year’s college student at Harbin Normal University, China, since 2009. She has been engaged in software architecture testing and software metrics research for approximately three years.

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