C2-Style Architecture Testing and Metrics Using Dependency Analysis

Lijun Lun
College of Computer Science and Information Engineering, Harbin Normal University, Harbin, China
Email: lunlijun@yahoo.cn

Xin Chi
College of Computer Science and Information Engineering, Harbin Normal University, Harbin, China
Email: chixin9010@yahoo.cn

Xuemei Ding
Faculty of Software, Fujian Normal University, Fuzhou, China
Email: dxmgw@yahoo.com.cn

***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 de 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 relationship between component and connector. Then the C2-style element dependency graph of C2-style architecture is constructed on the basis of these dependency relationships. Based on the model introduced, dependency edge coverage testing and dependency edge coverage metrics applications are given. And finally, discusses the equivalence between our methods and 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.

doi:10.4304/jsw.8.2.276-285

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 = (Comp, Conn), where:

- Comp = {$Comp_1$, $Comp_2$, …, $Comp_m$} is a finite set of components, where $Comp_i$ = {$Comp_i.I_{pt}\_i$, $Comp_i.I_{pt}\_o$, $Comp_i.I_{pb}\_i$, $Comp_i.I_{pb}\_o$}.
- Conn = {$Conn_1$, $Conn_2$, …, $Conn_n$} is a finite set of connectors, where $Conn_i$ = {$Conn_i.I_{nt}\_i_1$, $Conn_i.I_{nt}\_i_2$, …, $Conn_i.I_{nt}\_i_n$, $Conn_i.I_{nt}\_o_1$, $Conn_i.I_{nt}\_o_2$, …, $Conn_i.I_{nt}\_o_n$, $Conn_i.I_{nb}\_i_1$, $Conn_i.I_{nb}\_i_2$, …, $Conn_i.I_{nb}\_i_m$, $Conn_i.I_{nb}\_o_1$, $Conn_i.I_{nb}\_o_2$, …, $Conn_i.I_{nb}\_o_m$}.
- I_{pb}_i or I_{nb}_i is the set of requests received at the bottom side of component or connector. I_{pb}_o or I_{nb}_o is the set of notifications that component or connector emits from its bottom side.
- I_{pt}_i or I_{nt}_i is the set of notifications received on the top side of component or connector. I_{pt}_o or I_{nt}_o is the set of requests sent from its top side.

Fig. 1 represents the external view of a component $Comp_i$. $Comp_i.I_{pt}\_i$ and $Comp_i.I_{pt}\_o$ are defined by the component′s dialog. They are the requests it will be submitting and notifications it will be handling. $Comp_i.I_{pb}\_o$ is the notifications the component will be making, reflecting changes to its internal object. $Comp_i.I_{pb}\_i$ is the requests the component accepts.

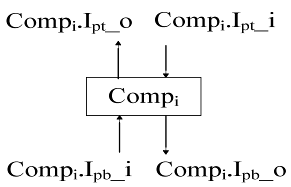

Figure 1. C2 component domains

Fig. 2 represents the external view of a connector $Conn_i$, with the components $Comp_{t_j}$ $(j = 1, ..., n)$ and $Comp_{c_k}$ $(k = 1, ..., m)$ attached to its top and bottom respectively. A connector's top and bottom domains of discourse are completely specified in terms of these components' interfaces. Therefore, a C2-style connector's interface is defined by the unions of the interfaces of the components above and below it, along with any filtering that the connector does to those interfaces. The interface will evolve dynamically as components are added, removed, and/or replaced. A connector′s top and bottom domain is completely specified in terms of these components.

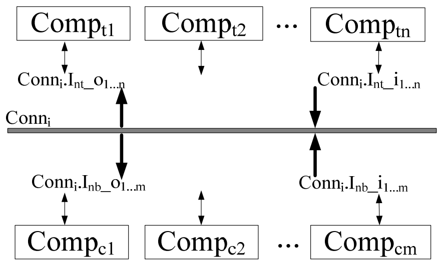

Figure 2. C2 connector domains

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 $Comp_1$ is a component in C2-style architecture, I_{pt} is the top of the $Comp_1$, $Conn_2$ is a connector in C2-style architecture, I_{nb} is the bottom of the $Conn_2$. If the change of $Comp_1.I_{pt}\_o$ affects $Conn_2.I_{nb}\_i$, then $Conn_2.I_{nb}\_i$ depends on $Comp_1.I_{pt}\_o$, denoted by $DEP_{pn}(Comp_1.I_{pt}\_o, Conn_2.I_{nb}\_i)$. Similar, $DEP_{pn}(Comp_1.I_{pb}\_o, Conn_2.I_{nt}\_i)$ represents $Conn_2.I_{nt}\_i$ depends on $Comp_1.I_{pb}\_o$.

Definition 2.3 Let $Comp_1$ is a component in C2-style architecture, I_{pb} is the bottom of the $Comp_1$, $Conn_2$ is a connector in C2-style architecture, I_{nt} is the top of the $Conn_2$. If the change of $Conn_2.I_{nt}\_o$ affects $Comp_1.I_{pb}\_i$, then $Comp_1.I_{pb}\_i$ depends on $Conn_2.I_{nt}\_o$, denoted by $DEP_{np}(Comp_1.I_{pb}\_i, Conn_2.I_{nt}\_o)$. Similar, $DEP_{np}(Comp_1.I_{pt}\_i, Conn_2.I_{nb}\_o)$ represents $Comp_1.I_{pt}\_i$ depends on $Conn_2.I_{nb}\_o$.

Definition 2.4 Let $Comp_1$ is a component in C2-style architecture, I_p is the interface of the $Comp_1$, $Conn_2$ is a connector in C2-style architecture, I_n is the interface of the $Conn_2$. If there are $DEP_{pn}(Comp_1.I_{pt}\_o, Conn_2.I_{nb}\_i)$ and $DEP_{np}(Comp_1.I_{pt}\_i, Conn_2.I_{nb}\_o)$, or $DEP_{pn}(Comp_1.I_{pb}\_o, Conn_2.I_{nt}\_i)$ and $DEP_{np}(Comp_1.I_{pb}\_i, Conn_2.I_{nt}\_o)$, then I_p and I_n depend on each other, denoted by $DEP_{I_{pn-b}}(I_p, I_n)$.

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

Property 1 $\forall (I_p, I_n)\ DEP_{I_{pn-b}}(I_p, I_n) \Rightarrow DEP_{pn}(Comp_1.I_p, Conn_2.I_n) \wedge DEP_{np}(Comp_1.I_p, Conn_2.I_n)$

Definition 2.5 Let $Conn_1$ and $Conn_2$ are two connectors in C2-style architecture, I_{nt} is the top of the $Conn_1$, I_{nb} is the bottom of the $Conn_2$. If the change of $Conn_1.I_{nt}\_o$ affects $Conn_2.I_{nb}\_i$, then $Conn_2.I_{nb}\_i$ depends on $Conn_1.I_{nt}\_o$, denoted by $DEP_{nn}(Conn_1.I_{nt}\_o, Conn_2.$

I_{nb}_i). Similar, DEP_{nn}($Conn_1.I_{nb}$_o, $Conn_2.I_{nt}$_i) represents $Conn_2.I_{nt}$_i depends on $Conn_1.I_{nb}$_o.

Definition 2.6 Let $Conn_1$ and $Conn_2$ are two connectors in C2-style architecture, I_{n_1} is the interface of the $Conn_1$, I_{n_2} is the interface of the $Conn_2$. If there are DEP_{nn}($Conn_1$. $I_{n_1^t}\_o$, $Conn_2$. $I_{n_2^b}\_i$) and DEP_{nn} ($Conn_2$. $I_{n_2^b}\_o$, $Conn_1$. $I_{n_1^t}\_i$), or DEP_{nn}($Conn_1$. $I_{n_1^b}\_o$, $Conn_2$. $I_{n_2^t}\_i$) and DEP_{nn}($Conn_2$. $I_{n_2^t}\_o$, $Conn_1$. $I_{n_1^b}\_i$), then I_{n_1} and I_{n_2} depend on each other, denoted by $DEP_{I_{nn}-b}(I_{n_1}, I_{n_2})$.

According to the definition 2.5 and 2.6, we have:

Property 2 $\forall(I_{n_1}, I_{n_2})\ DEP_{I_{nn}-b}(I_{n_1}, I_{n_2}) \Rightarrow$

$$DEP_{nn}(Conn_1.I_{n_1}, Conn_2.I_{n_2}) \wedge$$
$$DEP_{nn}(Conn_2.I_{n_2}, Conn_1.I_{n_1})$$

B. Dependency Relationship between Component and Connector

Definition 2.7 Let $Comp_1$ is a component in C2-style architecture, $Conn_2$ is a connector in C2-style architecture. If there is DEP_{pn}($Comp_1.I_{pt}$_o, $Conn_2.I_{nb}$_i) or DEP_{pn} ($Comp_1.I_{pb}$_o, $Conn_2.I_{nt}$_i), then $Conn_2$ depends on $Comp_1$, denoted by DEP_{pn}($Comp_1$, $Conn_2$).

Definition 2.8 Let $Comp_1$ is a component in C2-style architecture, $Conn_2$ is a connector in C2-style architecture. If there is DEP_{np}($Comp_1.I_{pb}$_i, $Conn_2.I_{nt}$_o) or DEP_{np} ($Comp_1.I_{pt}$_i, $Conn_2.I_{nb}$_o), then $Comp_1$ depends on $Conn_2$, denoted by DEP_{np}($Comp_1$, $Conn_2$).

Definition 2.9 Let $Comp_1$ is a component in C2-style architecture, I_p is the interface of the $Comp_1$, $Conn_2$ is a connector in C2-style architecture, I_n is the interface of the $Conn_2$. If there is $DEP_{I_{pn}-b}(I_p, I_n)$, then $Comp_1$ and $Conn_2$ depend on each other, denoted by $DEP_{pn\text{-}b}$($Comp_1$, $Conn_2$).

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

Property 3 $\forall(Comp_1, Conn_2)\ DEP_{pn-b}(Comp_1,$

$$Conn_2) \Rightarrow DEP_{pn}(Comp_1, Conn_2) \wedge$$
$$DEP_{np}(Comp_1, Conn_2)$$

C. Dependency Relationship between Connector

Definition 2.10 Let $Conn_1$ and $Conn_2$ are two connectors in C2-style architecture, I_{nt} is the interface of the $Conn_1$, I_{nb} is the interface of the $Conn_2$. If there are DEP_{nn}($Conn_1.I_{nt}$_o, $Conn_2.I_{nb}$_i) and DEP_{nn}($Conn_2.I_{nb}$_o, $Conn_1.I_{nt}$_i), then $Conn_2$ depends on $Conn_1$, denoted by DEP_{nn}($Conn_1$, $Conn_2$).

Definition 2.11 Let $Conn_1$ and $Conn_2$ are two connectors in C2-style architecture. If there are DEP_{nn} ($Conn_1$, $Conn_2$) and DEP_{nn}($Conn_2$, $Conn_1$), then $Conn_1$ and $Conn_2$ depend on each other, denoted by $DEP_{nn\text{-}b}$($Conn_1$, $Conn_2$).

According to the definition 2.10 and 2.11, we have:

Property 4 $\forall(Conn_1, Conn_2)\ DEP_{nn-b}(Conn_1,$

$$Conn_2) \Rightarrow DEP_{nn}(Conn_1, Conn_2) \wedge$$
$$DEP_{nn}(Conn_2, Conn_1)$$

D. Dependency Relationship in Component and Connector

Definition 2.12 Let $Comp_1$ is a component in C2-style architecture. If there is a bottom of $Comp_1$ depends on a top of $Comp_1$, or a top of $Comp_1$ depends on a bottom of $Comp_1$, denoted by DEP_p($Comp_1$).

Definition 2.13 Let $Conn_1$ is a connector in C2-style architecture. If there is a bottom of $Conn_1$ depends on a top of $Conn_1$, or a top of $Conn_1$ depends on a bottom of $Conn_1$, denoted by DEP_n($Conn_1$).

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 an C2-style architecture, the C2-style element dependency graph for the C2-style architecture denoted by C2-EDG = $<V, E>$, where:

- $V = <Comp, Conn, I_{pt}, I_{pb}, I_{nt}, I_{nb}>$.
- Comp represents the set of components in C2-style.
- Conn represents the set of connectors in C2-style.
- I_{pt} represents the set of top interfaces of component in C2-style. $Comp_i.I_{pt}$_o represents the set of requests sent from its top side of a component $Comp_i$. $Comp_i.I_{pt}$_i represents the set of notifications received on the top side of a component $Comp_i$.
- I_{pb} represents the set of bottom interfaces of component in C2-style. $Comp_i.I_{pb}$_o represents the set of requests sent from its bottom side of a component $Comp_i$. $Comp_i.I_{pb}$_i represents the set of notifications received on the bottom side of a component $Comp_i$.
- I_{nt} represents the set of top interfaces of connector in C2-style. $Conn_j.I_{nt}$_o represents the set of requests sent from its top side of a connector $Conn_j$. $Conn_j.I_{nt}$_i represents the set of notifications received on the top side of a connector $Conn_j$.
- I_{nb} represents the set of bottom interfaces of connector in C2-style. $Conn_j.I_{nb}$_o represents the set of requests sent from its bottom side of a connector $Conn_j$. $Conn_j.I_{nb}$_i represents the set of notifications received on the bottom side of a connector $Conn_j$.
- $E = \{<Comp_1.I_{pt}\_o, Conn_2.I_{nb}\_i> \vee <Conn_1.I_{pt}\_o, Comp_2.I_{nb}\_i> \vee <Comp_1.I_{pb}\_o, Conn_2.I_{no}\_i> \vee <Conn_1.I_{pb}\_o, Comp_2.I_{nt}\_i> \vee < Conn_1^{'}.I_{nt}\_o, Conn_2^{'}.I_{nb}\_i > \vee < Conn_1^{'}.I_{nb}\_o, Conn_2^{'}.I_{nt}\_i > \vee <Comp_1.I_{pt}\_o, Comp_1.I_{pb}\_i> \vee <Comp_1.I_{pb}\_o, Comp_1.I_{pt}\_i> \vee <Conn_2.I_{pt}\_o, Conn_2.I_{pb}\_i> \vee <Conn_2.I_{pb}\_o, Conn_2.I_{pt}\_i> \mid (Comp_1, Comp_2 \in Comp \wedge Conn_1, Conn_2, Conn_1^{'}, Conn_2^{'} \in Conn)$

$\vee$ ((DEP$_{pn}$(Comp$_1$, Conn$_2$) $\vee$ DEP$_{np}$(Comp$_1$, Conn$_2$) $\vee$ DEP$_{nn}$($Conn'_1$, $Conn'_2$) $\vee$ DEP$_p$ (Comp$_1$) $\vee$ DEP$_n$(Conn$_2$))} represents the set of edges.

The C2-EDG of C2-style architecture can be constructed using the following steps:

- The architecture of each component and connector, an increase in the corresponding node.
- If the interface I_n of connector Conn$_2$ depends on the interface I_p of component Comp$_1$, then there is a dependency edge from I_p to I_n.
- If the interface I_p of component Comp$_1$ depends on the interface I_n of connector Conn$_2$, then there is a dependency edge from I_n to I_p.
- If the interface I_{n_2} of connector Conn$_2$ depends on the interface I_{n_1} of connector Conn$_1$, then there is a dependency edge from I_{n_1} to I_{n_2}.
- If I_{p_1} and I_{p_2} are two interfaces of component Comp$_1$, I_{p_2} depends on I_{p_1}, then there is a additional dependency edge from I_{p_1} to I_{p_2} in Comp$_1$.
- If I_{n_1} and I_{n_2} are two interfaces of connector Conn$_2$, I_{n_1} depends on I_{n_2}, then there is a additional dependency edge from I_{n_2} to I_{n_1} in Conn$_2$.

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

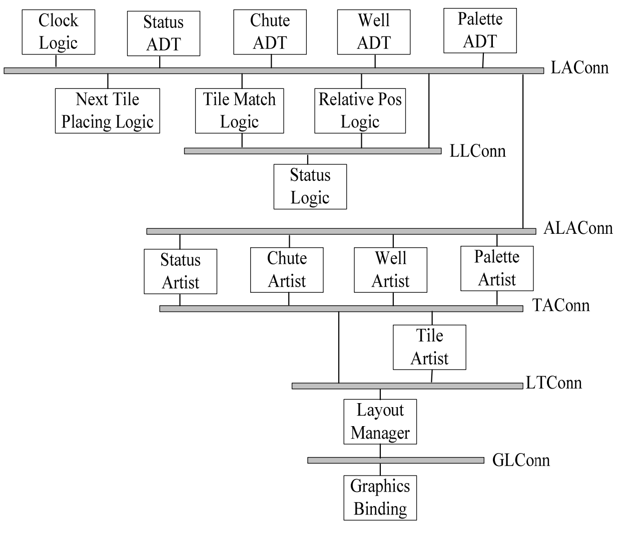

Figure 3. KLAX architecture in the C2-style

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 component to an interface of a corresponding connector. Thick dashed edge represents dependency edge from connector to component that connected an interface of a connector and an interface of a corresponding component. Thick dotted 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.I_{pt}_o, LTConn.I_{nb}_i) represents the dependency edge from component LayoutManager to connector LTConn, (ALAConn.I_{nb}_o, PaletteADT.I_{pt}_i) represents the dependency edge from connector ALAConn to component PaletteADT, and (LTConn.I_{nt}_o, TAConn.I_{nb}_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}.

I_{pt} = {GraphicsBinding.I_{pt}_o, GraphicsBinding.I_{pt}_i, LayoutManager.I_{pt}_o, LayoutManager.I_{pt}_i, TileArtist.I_{pt}_o, TileArtist.I_{pt}_i, StatusArtist.I_{pt}_o, StatusArtist.I_{pt}_i, …}.

I_{pb} = {LayoutManager.I_{pb}_o, LayoutManager.I_{pb}_i, TileArtist.I_{pb}_o, TileArtist.I_{pb}_i, StatusArtist.I_{pb}_o, Status Artist.I_{pb}_i, ChuteArtist.I_{pb}_o, ChuteArtist. I_{pb}_ i, …}.

I_{nt} = {GLConn.I_{nt}_o, GLConn.I_{nt}_i, LTConn.I_{nt}_o, LTConn.I_{nt}_i, TAConn.I_{nt}_o, TAConn.I_{nt}_i, ALAConn.I_{nt}_o, ALAConn.I_{nt}_i, LAConn.I_{nt}_o, LAConn.I_{nt}_i, LLConn.I_{nt}_o, LLConn.I_{nt}_i}.

I_{nb} = {GLConn.I_{nb}_o, GLConn.I_{nb}_i, LTConn.I_{nb}_o, LTConn.I_{nb}_i, TAConn.I_{nb}_o, TAConn.I_{nb}_i, ALAConn.I_{nb}_o, ALAConn.I_{nb}_i, LAConn.I_{nb}_o, LAConn.I_{nb}_i, LLConn.I_{nb}_o, LLConn.I_{nb}_i}.

E = {<GraphicsBinding.I_{pt}_o, GLConn.I_{nb}_i>, <GL Conn.I_{nb}_o, GraphicsBinding.I_{pt}_i>, <GLConn.I_{nt}_o, Lay outManager.I_{pb}_i>, …}.

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 inspection, which are closely related to the software architecture testing and the traditional testing.

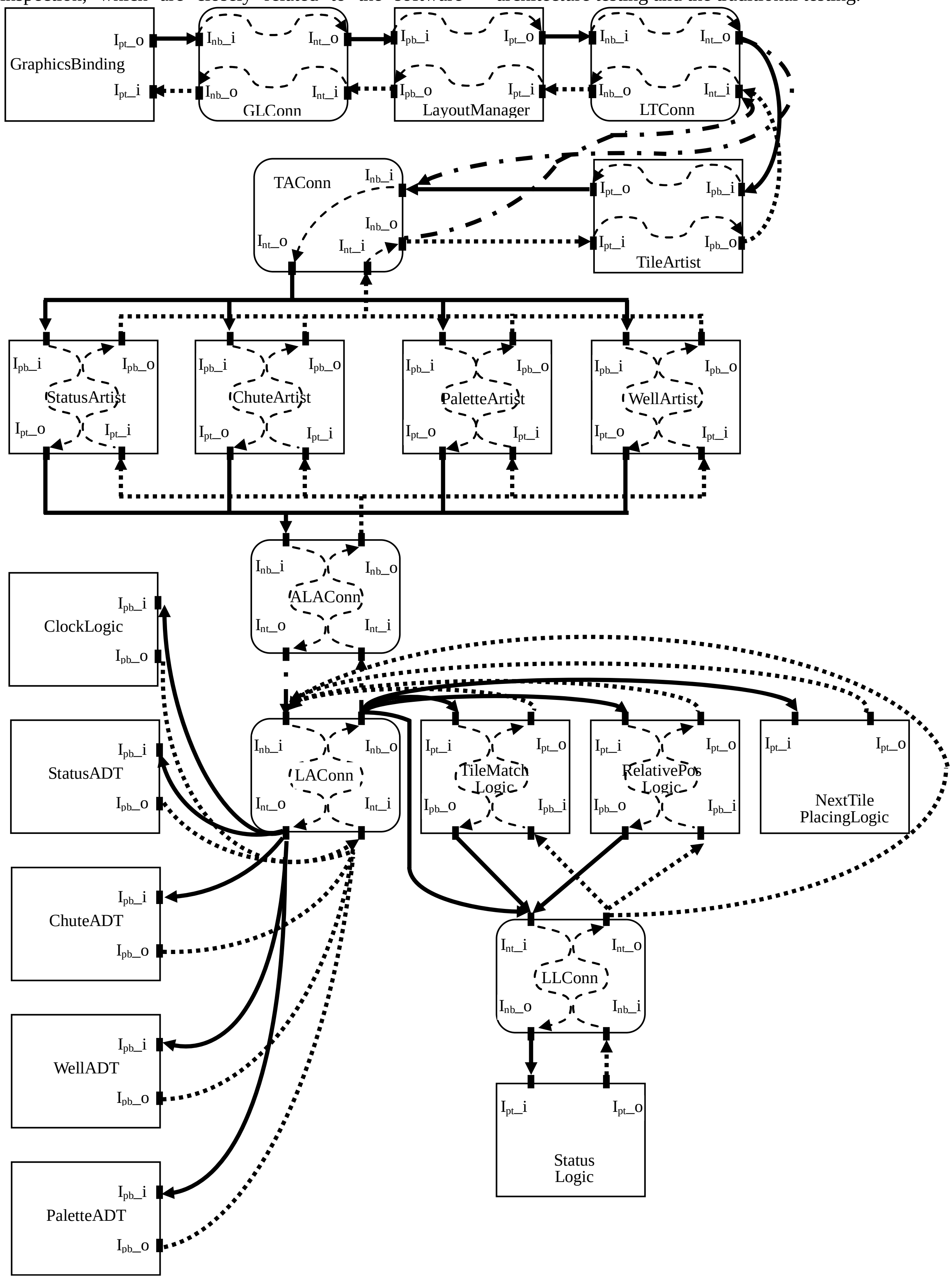

Figure 4. C2-EDG of Fig. 3

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 $e_{V(i),V(j)}$ represents the dependency edge of C2-EDG, where $V(i)$ and $V(j)$ are nodes of C2-EDG, and $TS_{e_{V(i),V(j)}}$ represents a set of test cases created to satisfy $e_{V(i),V(j)}$.

(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 $e_{Comp_1,Conn_2}$ in C2-EDG, there is at least one test case $t_{Comp_1,Conn_2} \in TS_{e_{Comp_1,Conn_2}}$ such that there is a DEP_{pn} $(Comp_1, Conn_2)$ induced by $e_{Comp_1,Conn_2}$, 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:

< $Comp_i.I_{pt}$_o, $Conn_j.I_{nb}$_i > ## or

< $Comp_i.I_{pb}$_o, $Conn_j.I_{nt}$_i >

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

The dependency edge coverage criteria for connector to component requires that every $DEP_{np}(Comp_1, Conn_2)$ in C2-EDG be covered by at least one test case.

Definition 3.2 For every dependency edge $e_{Conn_2,Comp_1}$ in C2-EDG, there is at least one test case $t_{Conn_2,Comp_1} \in TS_{e_{Conn_2,Comp_1}}$ such that there is a DEP_{np} $(Comp_1, Conn_2)$ induced by $e_{Conn_2,Comp_1}$, 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:

< $Conn_i.I_{nb}$_o, $Comp_j.I_{pt}$_i > ## or

< $Conn_i.I_{nt}$_o, $Comp_j.I_{pb}$_i >

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

The dependency edge coverage criteria for connector to connector requires that every $DEP_{nn}(Conn_1, Conn_2)$ in C2-EDG be covered by at least one test case.

Definition 3.3 For every dependency edge $e_{Conn_1,Conn_2}$ in C2-EDG, there is at least one test case $t_{Conn_1,Conn_2} \in TS_{e_{Conn_1,Conn_2}}$ such that there is a DEP_{nn} $(Conn_1, Conn_2)$ induced by $e_{Conn_1,Conn_2}$, 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:

< $Conn_i.I_{nt}$_o, $Conn_j.I_{nb}$_i > ## or

< $Conn_i.I_{nb}$_o, $Conn_j.I_{nt}$_i >

To verify the C2-style, we carry out experiment [14] on the KLAX system. Tab. I show 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

Name of Coverage Criteria	Number of Dependency Edges
DEComp_ConnCC	24
DEConn_CompCC	24
DEConn_ConnCC	6

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 DEConn-CompCC, 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 DEComp-ConnCC.

Thus, this concludes the proof.

TABLE II.

DEPENDENCY COVERAGE RESULT FOR EXPERIMENT

Connector Name	Connector Name					
	GL Conn	***LT Conn***	***TA Conn***	***ALA Conn***	***LL Conn***	***LA Conn***
GLConn	No	No	No	No	No	No
LTConn	No	No	Yes	No	No	No
TAConn	No	Yes	No	No	No	No
ALAConn	No	No	No	No	No	Yes
LLConn	No	No	No	No	No	Yes
LAConn	No	No	No	Yes	Yes	No

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 $\| Comp \|$ is number of components of C2-style architecture, $\| Conn \|$ is number of connectors of C2-style architecture, $\| e_{Comp,Conn} \|$ is the number of dependency edges from component to connector, $\| e_{Conn,Comp} \|$ is the number of dependency edges from connector to component, $\| e_{Conn,Conn} \|$ 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:

$$DEC_{Comp}^{Conn} = \frac{\sum_{i=1}^{\|Comp\|} \sum_{j=1}^{\|Conn\|} \| e_{Comp_i, Conn_j} \|}{\| Comp \| + 2 \| Conn \|} \times 100\% \quad (1)$$

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:

$$DEC_{Conn}^{Comp} = \frac{\sum_{i=1}^{\|Conn\|} \sum_{j=1}^{\|Comp\|} \| e_{Conn_i, Comp_j} \|}{\| Comp \| + 2 \| Conn \|} \times 100\% \quad (2)$$

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:

$$DEC_{Conn}^{Conn} = \frac{\sum_{i=1}^{\|Conn\|} \sum_{j=1}^{\|Conn\|} \| e_{Conn_i, Conn_j} \|}{\| Comp \| + 2 \| Conn \|} \times 100\% \quad (3)$$

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:

$$DEC^{C2} = \frac{DEC_{Comp}^{Conn} + DEC_{Conn}^{Comp} + DEC_{Conn}^{Conn}}{3} \quad (4)$$

Tab. 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:

$$DEC^{C2} = \frac{1}{3}(85.7\% + 85.7\% + 21.4\%) = 64.3\%$$

TABLE III.

DEPENDENCY TEST COVERAGE IN KLAX

Coverage	Computation	Result
DEC_{Comp}^{Conn}	$= \frac{8 \times 1 + 8 \times 2}{16 + 2 \times 6}$	= 85.7 %
DEC_{Conn}^{Comp}	$= \frac{2 \times 2 + 5 + 4 + 3 + 8}{16 + 2 \times 6}$	= 85.7 %
DEC_{Conn}^{Conn}	$= \frac{1 + 1 + 1 + 1 + 1 + 1}{16 + 2 \times 6}$	= 21.4 %

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.interf$_1$, N.interf$_2$), Component(Connector)_Internal_Sequencing_Relation(N.interf$_1$, N.interf$_2$), Component(Connector)_Internal_Relation (N$_1$.interf$_1$, N$_1$.interf$_2$), N_C_Relation(N.interf$_1$, C.interf$_1$) or C_N_Relation(C.interf$_1$, N.interf$_1$), Direct_Component_Relation(N$_1$.interf$_1$, C$_1$.interf$_1$, C$_1$.interf$_2$, N$_2$.interf$_2$), and Indirect_Component_Relation(N$_1$.interf$_1$, C$_1$.interf$_1$, C$_1$.interf$_2$, N$_2$.interf$_2$, C$_2$.interf$_1$, C$_2$.interf$_2$, N$_3$.interf$_1$). 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$_i$ is component, Conn$_j$ is connector, and interf$_k$ is interface. Where:

- DEP$_{pn}$(Comp.I$_{pt}$_o, Conn.I$_{nb}$_i) or DEP$_{pn}$(Comp.I$_{pb}$_o, Conn.I$_{nt}$_i) is equivalent to Comp_Conn_ Relation(Comp.interf$_i$, Conn.interf$_j$).
- DEP$_{np}$(Comp.I$_{pt}$_i, Conn.I$_{nb}$_o) or DEP$_{np}$(Comp.I$_{pb}$_i, Conn.I$_{nt}$_o) is equivalent to Conn_Comp_ Relation(Conn.interf$_i$, Comp.interf$_j$).
- DEP$_{pn}$(Comp$_1$, Conn$_2$) and DEP$_{np}$(Comp$_3$, Conn$_2$) is equivalent to Direct_Component_Relation (Comp$_1$.interf$_i$, Conn$_2$.interf$_j$) and Conn_Comp_ Relation(Conn$_2$.interf$_k$, Comp$_3$.Interf$_l$).

B. *Our Methods are Equivalent to Gao′Method*

Gao et al. proposed an adequate test model [15], known as a CFAGs and D-CFAGs, 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 $Conn_j$ is connector. Where:

- Node coverage criterion and all-node-coverage criterion and is equivalent to $<Comp.I_{pt}\_o, Conn.I_{nb}\_i>$ or $<Comp.I_{pb}\_o, Conn.I_{nt}\_i>$ or $<Conn.I_{nb}\_i, Comp.I_{pt}\_o>$ or $<Conn.I_{nt}\_i, Comp.I_{pb}\_o>$.
- Link coverage criterion and all-link coverage criterion is equivalent to the combination of $<Comp_1.I_{pt}\_o, Conn_2.I_{nb}\_i>$, $<Conn_2.I_{nt}\_o, Conn_3.I_{nb}\_i>$, …, $<Conn_i.I_{nt}\_o, Conn_{i+1}.I_{nb}\_i>$, and $<Conn_{i+1}.I_{nt}\_o, Comp_3.I_{pb}\_i>$ or $<Comp_1.I_{pb}\_o, Conn_2.I_{nt}\_i>$, $<Conn_2.I_{nb}\_o, Conn_3.I_{nt}\_i>$, …, $<Conn_i.I_{nb}\_o, Conn_{i+1}.I_{nt}\_i>$, and $<Conn_{i+1}.I_{nb}\_o, Comp_3.I_{pt}\_i>$.
- If there are relations that connect $Comp_1$, $Comp_2$, $Comp_3$, $Conn_1$, and $Conn_2$ together, then the result path $Comp_1 - Conn_1 - Comp_2 - Conn_2 - Comp_3$ of path coverage criterion is equivalent to a number of combinations of $<Comp_1.I_{pt}\_o, Conn_1.I_{nb}\_i>$, $<Conn_1.I_{nt}\_o, Comp_2.I_{pb}\_i>$, $<Comp_2.I_{pt}\_o, Conn_2.I_{nb}\_i>$, and $<Conn_2.I_{nt}\_o, Comp_3.I_{pb}\_i>$ or $<Comp_1.I_{pb}\_o, Conn_1.I_{nt}\_i>$, $<Conn_1.I_{nb}\_o, Comp_2.I_{pt}\_i>$, $<Comp_2.I_{pb}\_o, Conn_2.I_{nt}\_i>$, and $<Conn_2.I_{nb}\_o, Comp_3.I_{pt}\_i>$.
- Minimum-set path coverage criterion is equivalent to the minimum of the length of path obtained from component $Comp_1$ to $Comp_2$ ($Comp_1 \neq Comp_2$) 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 $Path_k$, $len(Path_k) = \sum_{C' \in (Comp_i \vee Conn_j) \wedge Path_k} C'$.

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

[1] J. A. Stafford, J. D. McGregor, "Top-Down Analysis for Bottom-Up Development", *in: Proc. of the 2004 Workshop on Component-Oriented Programming (WCOP2004)*, Oslo, Norway, June 2004.

[2] F. Kon, R. H. Campbell, "Dependence Management in Component-Based Distributed Systems", *IEEE Concurrency*, vol. 8, pp. 26-36, January 2000.

[3] N. T. Richard, N. Medvidovic, K. M. Anderson, E. J. Whitehead, J. E. Robbins, "A Component- and Message-Based Architecture Style for GUI Software", *IEEE Trans. on Software Engi.*, vol. 22, pp. 390-406, June 1996.

[4] M. M. Gorlick, R. R. Razouk, "Using Weaves for Software Construction and Analysis", *in: Proc. of 13th Int'1. Conf. Software Engineering (ICSE1991)*, Austin, USA, pp. 23-34, May 1991.

[5] H. Muccini, A. Bertolino, P. Inverardi, "Using Software Architecture for Code Testing", *IEEE Trans. on Software Engi.*, vol. 30, pp. 160-171, March 2004.

[6] H. Muccini, M. Dias, D. J. Richardson, "Towards Software Architecture-based Regression Testing", *ACM SIGSOFT Software Engineering Notes*, vol. 30, pp. 1-7, April. 2005.

[7] L. J. Lun, H. Xu, "An Approach to Software Architecture Testing", *in Proc. of the 9th International Conference for Young Computer Scientists (ICYCS2008)*, Zhangjiajie, China, pp. 1070-1075, November 2008.

[8] L. J. Lun, X. M. Ding, "Analyzing Relation between Software Architecture Testing Criteria on Test Sequences", *in: Proc. of 2009 IEEE Secure Software Integration and Reliability Improvement (SSIRI2009)*, Shanghai, China, pp. 181-186, July 2009.

[9] J. Zhenyi, J. Offutt, "Deriving Tests from Software Architectures", *in Proc. of 12th IEEE International Symposium on Software Reliability Engineering (ISSRE2001)*, Hong Kong, China, pp. 308-313, November 2001.

[10] A. Bertolino, F. Corradini, P. Inverardi, H. Muccini, "Deriving Test Plans from Architectural Descriptions", *in: ACM Proceedings International Conference on Software Engineering (ICSE2000)*, Limerick, Ireland, pp. 220-229, June 2000.

[11] M. Shereshevsky, H. Ammari, N. Gradetsky, A. Mili, H. H. Ammar, "Information Theoretic Metrics for Software Architectures", in: *Proc. of the 25th International Computer Software and Applications Conference on Invigorating Software Development (COMPSAC2001)*, Chicago, IL, USA, pp. 151-157, October 2001.

[12] J. Zhao, H. Yang, L. Xiang, B. Xu, "Change impact analysis to support architectural evolution", *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 14, pp. 317-333, May 2002.

[13] H. Muccini, M. Dias, D. J. Richardson, "Systematic Testing of Software Architectures in the C2 style", *Lecture Notes in Computer Science*, vol. 2984, pp. 295-309, 2004.

[14] L. J. Lun, X. Chi, "Software Architecture Testing in the C2 Style", *in: Proc. of 2010 3rd International Conference on Advanced Computer Theory and Engineering (ICACTE2010)*, Chengdu, China, vol. 1, pp. 123-127, August 2010.

[15] J. Gao, R. Espinoza, J. He, "Testing Coverage Analysis for Software Component Validation", *in: Proc. of the 29th Annual International Computer Software and Applications Conference (COMPSAC2005)*, Edinburgh, UK, vol. 1, pp. 463-470, July 2005.

[16] S. Horwitz, T. Reps, D. Binkley, "Interprocedural Slicing Using Dependence Graphs", *ACM Transactions on Programming Languages and Systems*, vol. 12, pp. 26-60, January 1990.

[17] A. Podgurski, L. A. Clarke, "A Formal Model of Program Dependences and its Implication for Software Testing, Debugging, and Maintenance", *IEEE Trans. on Software Engi.*, vol. 16, pp. 965-979, September 1990.

[18] J. Zhao, "A Slicing-based Approach to Extracting Reusable Software Architectures", *in: Proc. of the Fourth European Conference on Software Maintenance and Reengineering (CSMR2000)*, Zurich, Switzerland, pp. 215-223, February 2000.

[19] J. Zhao, "On Assessing the Complexity of Software Architectures", *in: Proc. of the third international workshop on Software architecture (ISAW1998)*, Orlando, USA, pp. 163-166, November 1998.

[20] J. A. Stafford, D. J. Richardson, A. L. Wolf, "Chaining: a software architecture dependence analysis technique", *Technical Report, CU 2CS2845297*, Department of Computer Science, University of Colorado, September 1997.

[21] J. A. Stafford, D. J. Richardson, A. L. Wolf, "Architectural-Level Dependence Analysis for Software Systems", *International Journal of Software Engineering and Knowledge Engineering*, vol. 1, pp. 431-451, August 2001.

[22] J. A. Stafford, A. L. Wolf, M. Caporuscio, "The Application of Dependence Analysis to Software Architecture Descriptions", *Lecture Notes in Computer Science*, vol. 2804, pp. 52-62, 2003.

[23] J. Zhao, "Using Dependence Analysis to Support Software Architecture Understanding", *New Technologies on Computer Software*, pp. 135-142, September, 1997.

[24] N. L. Hashim, S. Ramakrishnan, H. W. Schmidt, "Architectural Test Coverage for Component-based Integration Testing", *in: Proc. of seventh International Conference on Quality Software (QSIC2007)*, Portland, USA, pp. 262-267, October 2007.

[25] L. J. Lun, X. Chi, X. M. Ding, "Edge Coverage Analysis for Software Architecture Testing", *Journal of Software*, vol. 7, pp. 1121-1128, May 2012.

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.