Description and Verification of Dynamic Software Architectures for Distributed Systems

Hongzhen Xu[1,2,3]
[1] Department of computer Science and Technology, Tongji University, Shanghai, 201804, China
[2] Department of Computer Science and Technology, East China Institute of Technology, Fuzhou, Jiangxi Province, 344000, China
[3]Embedded System and Service Computing Key Lab of Ministry of Education, Shanghai, 201804, China
Email: xhz_97@163.com

Guosun Zeng[1,3] and Bo Chen[1,3]
[1] Department of computer Science and Technology, Tongji University, Shanghai, 201804, China
[3]Embedded System and Service Computing Key Lab of Ministry of Education, Shanghai, 201804, China
Email: gszeng@tongji.edu.cn, cb31@sina.com

***Abstract*—Continuing growth and increasing complexity of distributed software systems make them be more flexible, adaptive and easily extensible. Dynamic evolution or reconfiguration of distributed software systems is one possible solution to meet these demands. However, there are some challenges for building dynamically evolving distributed software systems at runtime, where dynamic software architectures for them is one of the most crucial problems. In this paper, we proposed a formal method of describing and verifying dynamic software architectures for distributed systems using hypergraph grammars. We firstly gave out reconfiguration production rules and operations for software architectures based on hypergraph grammars, and then described dynamic reconfiguration of software architectures for distributed systems according to those rules. At last we verified the invariant property of dynamic software architectures for those systems using model checking, and gave out corresponding verification algorithms.**

***Index Terms*—distributed system; dynamic software architectures; hypergraph grammars; invariant property; verification;**

I. INTRODUCTION

Today, enterprise information systems are typically distributed and have different requirements. In particular, as the Internet becomes a dominant runtime environment for software systems, its open and dynamic characteristics make user requirements and hardware resources change more rapidly, which make distributed software systems be more flexible, adaptive and easily extensible. For example, in large open distributed systems components may appear or disappear dynamically as the result of individual user's actions. Traditional approaches for software design and development cannot handle situations where client applications need evolve over time, and when the server application cannot be stop for maintenance. Dynamic evolution or reconfiguration of distributed software systems is one possible solution to meet these demands. Dynamic evolution of distributed software systems means that systems evolution occurs during execution of those systems, which can be implemented dynamically either by hot-swapping existing components or by integrating newly developed components without the need for stopping the system [1]. However, there are some challenges for building dynamically evolving distributed systems at runtime, while dynamic software architectures for them is one of the most crucial problems.

Software architecture researches focus on specification and description of software systems at a high-level [2]. A software architecture for a specific system describes a structure composed of components, connectors, and rules characterizing the interaction between these components and connectors [3]. In general, the software architecture acts as a bridge between requirements and implementation and provides a blueprint for system construction and composition. It helps to understand large systems, support reuse at architecture levels, expose the changeability of systems, verify the target system at a high level and so on [4].

Dynamic software architectures[5] are those architectures that modify their architecture and enact the modifications during the system's execution. This behavior is most commonly known as run-time *evolution* or *reconfiguration*. Dynamic software architectures model a software system that reacts to certain events at runtime by supporting reconfiguration of the system's architectures. The typical reconfiguration operations for dynamic software architectures include adding components or connectors, removing components or connectors, updating components or connectors, changing the architecture topology by adding or removing connections between components and connectors. Due to the advantage of continuous availability, the research on dynamic software architectures has become a hot issue in software engineering field.

In this paper, we focus on describing and verifying dynamic software architectures for distributed systems. We use hypergraph grammars as a formal framework for describing dynamic software architectures for distributed systems, and verify the invariant property of them with model checking. We select hypergraph grammars mainly because: (i) It provides both a graphical representation and a formal basis that is in line with the usual way architectures are represented, (ii) It allows us to rigorously reason about and verify architectural properties during dynamic evolution of distributed systems.

This paper is organized as follows. Section 2 gives a survey on the related work. Section 3 introduces our notational basis of hypergraph grammars. Section 4 demonstrates our approach to describing dynamic software architectures for distributed systems using hypergraph grammars. Section 5 shows how to verify the invariant property of dynamic software architectures for distributed systems using model checking. The last section concludes our approach and proposes future research.

II. Related Works

Many research works are focusing on describing, analyzing software architectures and their dynamic reconfiguration. They can be sub-divided into three categories. The first uses ADLs (Architectural Description Languages) to model and analyze software architectures and their reconfiguration. Although there are a large number of such languages, only a few are capable of modeling dynamic software architectures. Darwin[6] is solely concerned with the structural aspects of the software architecture, and lack description of how reconfigurations interact with on-going computations. Wright[7] is a static ADL, and make use of CSP(communication sequence process) to specify software architecture evolution. Dynamic Wright[8] is an expansion to Wright. It can describe dynamic reconfiguration of software architectures through a configurator. LEDA[9] uses the π-calculus to describe software architecture evolution, and cannot provide the basis of formal analysis for dynamic behavioral. π-ADL[10] is a architecture description language based on high-order calculus which can describe static, dynamic and mobile architecture from the behavioral and structural views. However, most of those ADLs do not offer graphical tools to display these models.

The second uses informal notations such as UML (Unified Modeling Language) and its extended models[11,12] to design software architectures and their reconfiguration. UML is expressive and easy to understand for its set of graphical notations, and has great improvement in UML 2.0, however, it is generally criticized for the lack of means for verifying and validating the designed models. It is still not very well suitable to describe dynamic software architectures[13].

The third uses formal techniques such as graph based techniques[14-16], logic based[17-18] and algebra based[19-21]. D. L. Métayer[14] proposed to describe software architecture styles using graph grammars. M. Wermelinger and M. Endler et al.[15,17] aimed at providing specification languages for the software architecture and its evolution. R. Bruni et al.[16] focused on graphical representation of dynamic software architectures, and had no formal description. N. Aguirre and C. Canal et al.[18,19] described local evolution of software architectures without from global perspective. Wang Yinghui et al.[20,21] researched about model and ripple effect analysis of software architecture evolution based on reachability matrix. None of them provided formal verification of dynamic software architectures, and those methods based on logic and algebra cannot offer any graphical display to these models.

In our approach we provide both graphical notations and formal notations for describing dynamic software architectures for distributed systems, and we also provide formal model verification based on those notations.

III. Hypergraph Grammars

Definition 3.1 (Hyperedge) The hyperedge is an item that has a number of tentacles that are connected to nodes. A hyperedge is drawn as shown in Figure 1, which has *n* (for example *n=3*) tentacles, and labels denote connections between hyperedges and nodes.

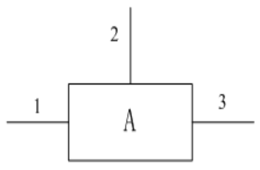

Figure 1. Hyperedge

Definition 3.2 (Hypergraph) A hypergraph is a tuple $\boldsymbol{H}= (\boldsymbol{V}, \boldsymbol{E}, \boldsymbol{s}, \boldsymbol{t}, \boldsymbol{l_V}, \boldsymbol{l_E})$, where

- $\boldsymbol{V}$ and $\boldsymbol{E}$ are disjoint finite sets of nodes and hyperedges, respectively.
- $\boldsymbol{s}, \boldsymbol{t} : \boldsymbol{E} \rightarrow \boldsymbol{V}^*$ are two mappings indicating the source nodes and the target nodes of a hyperedge, where each heperedge can be connected to a list of nodes.
- $\boldsymbol{l_V}$: $\boldsymbol{V} \rightarrow \Sigma_{\boldsymbol{V}}$, l_E: $\boldsymbol{E} \rightarrow \Sigma_{\boldsymbol{E}}$ are two label functions of hyperedges and nodes which are mappings from $\boldsymbol{V}$ and $\boldsymbol{E}$ in two finite sets of labels.

In our paper, we model components and connectors of distributed systems as hyperedges, communication ports between components and connectors as nodes. Nodes are represented by black dots. Component hyperedges are drawn as rectangles while connector hyperedges as rounded boxes, which are connected to their attached nodes by thin lines which carry the attachment labels. Figure 2 depicts the hypergraph of a software architecture for a distributed system which contains five components: *Web User*, *Mobile User*, *Mobile Gateway*, *Web Server* and *Database*; four connectors: *WConnector*, *GConnector* and *DConnector;* eight nodes *Pw*, *Pm*, *Ps*, *Pmg*, *Pg*, *Pgs*, *Psd* and *Pd*, where those nodes mean communication ports between components and connectors, *CR*、*CA*、*SR*、*SA* represent Client Request, Client Answer, Server Request and Server Answer respectively.

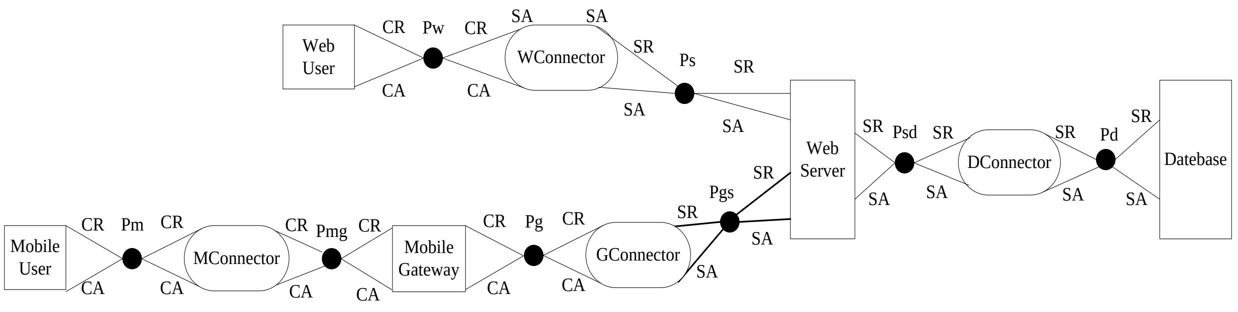

Figure 2. the Hypergraph of a software Architecture for a distributed system

Formally, a graph is a set of relation tuples noted *R(e$_1$, …, e$_n$)* where *R* is a *n*-ary relation name and *e$_i$* are entity names. We consider only binary and unary relations in this paper. We use a unary relation *U(e)* to characterize an entity *e*, and a binary relation *B(e$_1$, e$_2$)* to represent a link of name *B* between *e$_1$* and *e$_2$* in the architecture. For example, in Figure 2, we use *C(web user)*, *S(server)*, *NC(wconnector)* to represent a web client *web user*, a server *server* and a connector *wconnector* respectively. *Pw(web user, wconnector)*means a communication port between *web user* and *wconnector*. *CR(web user, wconnector)* corresponds to a client request from *web user* to *wconnector*, and *CA(wconnector, web user)* corresponds to a client answer from *wconnector* to *web user*, and so on. The architecture represented by Figure 2 is formally defined as the following set,

{ *C(web user), C(mobile user), G(mobile gateway), S(server), D(database), NC(wconnector), NC(mconnector), NC(gconnector), NC(dconnector), Pw(web user, wconnector),…, CR(web user, wconnector), CA(wconnector, web user),…* }

Definition 3.3 (Hypergraph Production Rule) A hypergraph production rule ***p*** = (***L***, ***R***), commonly written ***p***: ***L***→***R***, is a partial hypergraph morphism from ***L*** to ***R***, where ***L*** and ***R*** called left-hand side and right-hand side respectively.

Hypergraph production rules can be used to describe dynamic reconfiguration of software architectures for distributed systems. Given a software architecture for a distributed system, and its corresponding hypergraph is *H*, and given a hypergraph production rule *p:L→R*, the reconfiguration process can be informally described as follow: (1) find a match of the left-hand-side *L* in *H*, i.e., identify a subgraph of *H* that corresponds with *L*; (2) remove all the items corresponding to the left-hand side *L* but not in the right-hand-side *R* from the hypergraph *H*; (3) add all the items of *R* that are not in L; (4) the elements that are both in *L* and *R* are preserved ; (5) then we get the new hypergraph *H'* from the evolution of *H*, which means we get the hypergraph *H'* of the reconfigured software architecture.

Figure 3 shows the reconfiguration process of a software architecture using a hypergraph production rule. In the following of this paper, we also call hypergraph production rules as reconfiguration production rules.

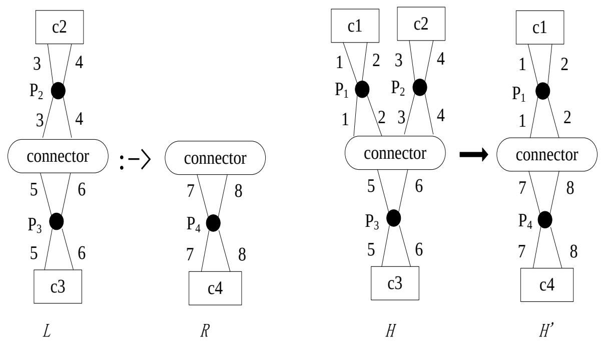

Figure 3. Reconfiguration of a Software Architecture Using a Hypergraph Production Rule

Definition 3.4 (Hypergraph Grammar) A hypergraph grammar is a tuple **G** = (***H***, ***P***, *H$_0$*), where ***H*** is a set of hypergraphes, ***P*** is a set of hypergraph production rules and *H$_0$* is the initial hypergraph.

We write $H \xrightarrow[p]{} H'$ to denote that *H* is evolved in one step to *H'* by using the production rule *p*∈*P*. We abbreviate the reduction sequence $H_0 \xrightarrow[p_1]{} H_1 \xrightarrow[p_2]{} H_2 \rightarrow \cdots \xrightarrow[p_n]{} H_n$ with $H \xrightarrow[p_1 p_2 \cdots p_n]{} H'$. We write $H \xrightarrow[*]{} H'$ to denote that there exists a sequence (possible empty) *s* of derivation steps such that $H \xrightarrow[s]{} H'$.

Using hypergraph grammars, we can formally describe the dynamic reconfiguration process of software architectures for distributed systems, and we can verify some properties of dynamic software architectures for those systems using formal methods such as modeling checking.

IV. Description of Dynamic Software Architectures for Distributed Systems

A. *Reconfiguration Production Rules and Operations*

A variety of definitions of dynamicity for software architectures have been proposed in the literature. Below we list two of the most prominent definitions to show the variability of connotations that the word dynamic acquires.

- **Programmed Dynamism**[16,17,22]: All admissible changes or reconfiguration for software architectures are defined prior to runtime and is triggered by the system when some conditions are satisfied.

- **Self-Repair Dynamism**[16,23,24]: Changes for software architectures are initiated and assessed internally or externally, i.e., the runtime behavior of the system is monitored to determine whether a reconfiguration is needed. In such case, a reconfiguration of software architectures is automatically performed.

In this paper, we only focus on the first dynamism, while the other dynamisms will be study in the future. In order to describe dynamic reconfiguration of software architectures for distributed systems, we predefine the following reconfiguration production rules and corresponding operations for dynamic software architectures for distributed systems before execution of systems.

1) Adding Production Rules and their Operations

If necessary, we can add new components or connectors to a software architecture for a distributed system at run-time. Supposing the current hypergraph of the software architecture is H_1, adding production rules for a new component and a new connector are following,

$$H_1 \to H_1 \cup \{C(c), R_1(c, connector), ..., P_1(c, connector), ...\} \quad (1)$$

$$H_1 \to H_1 \cup \{NC(cr), R_1(c, cr), ..., P_1(c, cr), ...)\} \quad (2)$$

When we add a new component or a new connector to a software architecture, we must add connections and communication ports responding to the new component or the new connector. In production rules (1) and (2), *C(c)* means a component entity *c*, *NC(cr)* means a connector entity *cr*, R_i means a interacting connection between components and connectors, and P_i means a communication port between components and connectors. For example, Figure 4 shows the reconfiguration operation for adding a new component *c1*.

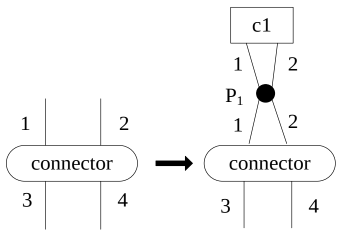

Figure 4. Reconfiguration Operation for Adding a new component

2) Removing Production Rules and their Operations

If a component or a connector is not being used by a software architecture, it can be removed. Removing production rules for software architectures for distributed systems are following,

$$H_1 \to H_1 - \{C(c), R_1(c, connector), ..., P_1(c, connector), ...\} \quad (3)$$

$$H_1 \to H_1 - \{NC(cr), R_1(c, cr), ..., P_1(c, cr), ...)\} \quad (4)$$

When we remove a component or a connector from software architectures, we must remove connections and communication ports responding to the removed component or connector, For example, Figure 5 shows the reconfiguration operation for removing a component *c2*.

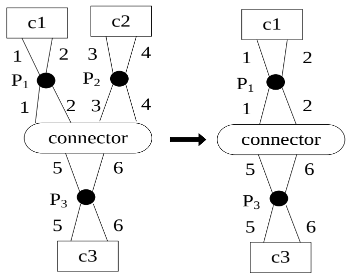

Figure 5. Reconfiguration Operation for Removing a component

3) Replacing Production Rules and their Operations

When necessary, a component or a connector of a software architecture for a distributed system can be replaced by another one with same signature, but with better performance. Replacing production rules for software architectures for distributed systems are following,

$$H_1 \to H_1 \cup \{C(c_1), R_1'(c_1, connector), ..., P_1'(c_1, connector), ...\} - \{C(c), R_1(c, connector), ..., P_1(c, connector), ...\} \quad (5)$$

$$H_1 \to H_1 \cup \{NC(cr_1), R_1'(c, cr_1), ..., P_1'(c, cr_1), ...)\} - \{NC(cr), R_1(c, cr), ..., P_1(c, cr), ...)\} \quad (6)$$

For example, Figure 6 shows the reconfiguration operation for replacing a connector *connector* with *connector1*.

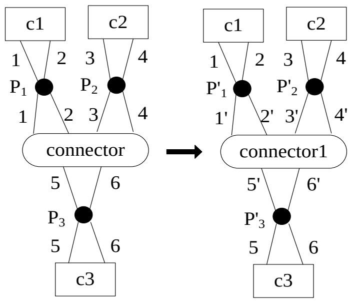

Figure 6. Reconfiguration Operation for Replacing a connector

B. A Case Study

1) A Case Scenario

In order to illustrate dynamic reconfiguration process of software architectures for distributed systems using hypergraph grammars, we introduce the following scenario of a web-based distributed application. Web clients and mobile clients can access web resources by making a request to a server group through connectors $WConnector_i$ and $MConnector_i$ respectively. The server group has only a control server *Control-Server* and many web servers WS_i. When the *Control-Server* receives access requests from clients, it is responsible for the distribution of requests to one of several geographically distributed web servers in the server group through the connector *SConnector*. Each web server has same functions, and maintains a queue of requests which are handled in FIFO order. Individual web server sends its response back to the requesting client through the *Control-Server*. The architecture is shown in Figure 7, where Pw_i and Ps_i are communication ports between web clients wc_i and *WConnector$_i$*, *Control-Server* and *WConnector$_i$*. *CR* and *CA* correspond to web client requests and web client answers. *SR* and *SA* correspond to the control server requests and the control server answers, and so on.

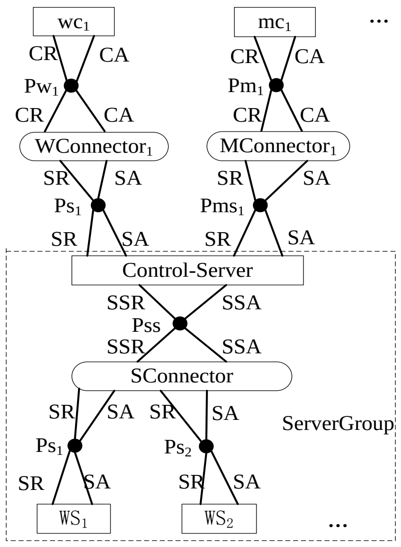

Figure 7. Example of a Distributed Software System

To improve the system efficiency and reduce operating costs, we need dynamically allocate the number of online web servers according to system operation. Let us assume that:

1) Each web server can accept at most *n* client requests (for the sake of simplicity, supposing *n=3*);
2) The system can activate or shutdown web servers in the server group when there are too many or too few client connection requests;
3) The system can retransmit a web client request waiting for answer from a web server to another web server when the first server is down.

2) Description of Dynamic Software Architectures

Supposing that there is no entity activated before the system initialization. After initialization, the system only activates the control server *Control-Server*. Let $H_0 = \varnothing$, which means the initial hypergraph of the system software architecture. When the system initializes, the configuration process is formally described as following when applying to the reconfiguration production rule (2),

$$H_0 = \varnothing \rightarrow \{S(cs)\} = H_1$$

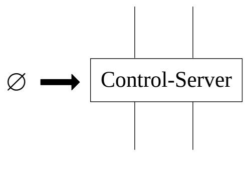

Figure 8. Initial Configuration of the System

Where we use *cs* to represent for the server *Control-Server*. We get the following hypergraph shown in Figure 8.

When a web client requires joining, the system adds in the web client, the corresponding connector, connections and communication ports by application to the reconfiguration production rule (1), (2). After then, if another mobile client asks to join successively, the system will adds in the new mobile client the corresponding connector, connections and communication ports. The reconfiguration process is formally following,

$$\begin{aligned}
H_1 \underset{2}{\rightarrow} H_1 &\cup \{NC(WConnector_1), SR(WConnector_1, cs),\\
&SA(cs, WConnector_1), Ps_1(WConnector_1, cs)\} = H_2\\
\underset{1}{\rightarrow} H_2 &\cup \{C(wc_1), CR(wc_1, WConnector_1), CA(WConnector_1, wc_1),\\
&Pw_1(wc_1, WConnector_1)\} = H_3\\
\underset{2}{\rightarrow} H_3 &\cup \{NC(SConnector), SSR(SConnector, cs),\\
&SSA(cs, SConnector), Pss(SConnector, cs)\} = H_4\\
\underset{1}{\rightarrow} H_4 &\cup \{S(ws_1), SSR(ws_1, SConnector), SSA(SConnector, ws_1),\\
&Ps_1(ws_1, SConnector)\} = H_5\\
\underset{2}{\rightarrow} H_3 &\cup \{NC(MConnector_1), SR(MConnector_1, cs),\\
&SA(cs, MConnector_1), Pms_1(MConnector_1, cs)\} = H_6\\
\underset{1}{\rightarrow} H_4 &\cup \{C(mc_1), CR(mc_1, MConnector_1), CA(MConnector_1, mc_1),\\
&Pm_1(mc_1, MConnector_1)\} = H_7
\end{aligned}$$

The reconfiguration process of software architectures for the distributed system is graphically shown in Figure 9.

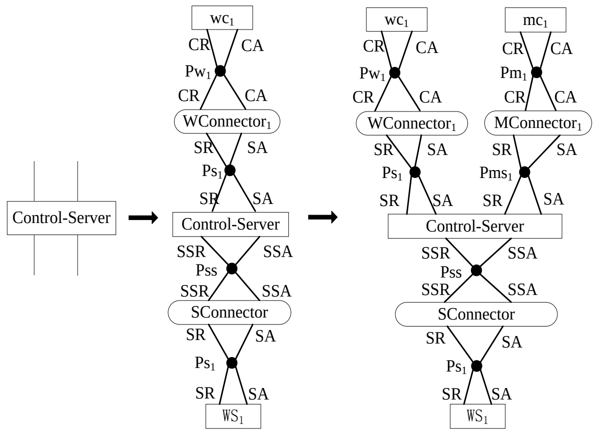

Figure 9. Dynamic Reconfiguration of System Architectures

When the web server WS_1 is down, the system will automatically replace WS_1 with another web server WS_1' by application to the reconfiguration production rule (5). The reconfiguration process of software architectures is formally following,

$$\begin{aligned}
H_7 \underset{5}{\rightarrow} H_7 &\cup \{S(ws_1'), SR'(SConnecort, ws_1'), SA'(ws_1', SConnecort),\\
&Ps_1'(ws_1', SConnecort)\} - \{S(ws_1'), SR(SConnecort, ws_1),\\
&SA(ws_1, SConnecort), Ps_1(ws_1, SConnecort)\} = H_8
\end{aligned}$$

The reconfiguration process of software architectures is graphically shown in Figure 10.

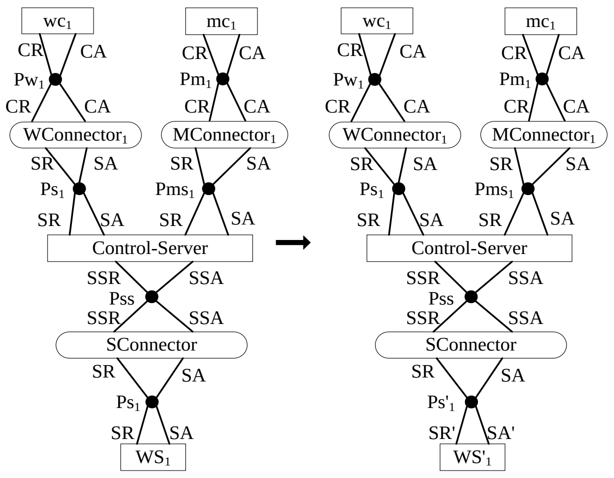

Figure 10. Dynamic Reconfiguration of Replacing for a Server

If two other web clients ask to join successively, the system will activate a new web server in the server group to provide services for the fourth web client according to the system assumption 1) and 2). The reconfiguration process is formally following,

$$
\begin{aligned}
&H_8 \underset{2}{\rightarrow} H_8 \cup \{NC(WConnector_2), SR(WConnector_2, cs),\\
&\quad SA(cs, WConnector_2), Ps_2(WConnector_2, cs)\} = H_9\\
&\underset{1}{\rightarrow} H_9 \cup \{C(wc_2), CR(wc_2, WConnector_2), CA(WConnector_2, wc_2),\\
&\quad Pw_2(wc_2, WConnector_2)\} = H_{10}\\
&\underset{2}{\rightarrow} H_{10} \cup \{NC(WConnector_3), SR(WConnector_3, cs),\\
&\quad SA(cs, WConnector_3), Ps_3(WConnector_3, cs)\} = H_{11}\\
&\underset{1}{\rightarrow} H_{11} \cup \{C(wc_3), CR(wc_3, WConnector_3), CA(WConnector_3, wc_3),\\
&\quad Pw_3(wc_3, WConnector_3)\} = H_{12}\\
&\underset{1}{\rightarrow} H_{12} \cup \{S(ws_2), SSR(ws_2, SConnector), SSA(SConnector, ws_2),\\
&\quad Ps_2(ws_2, SConnector)\} = H_{13}
\end{aligned}
$$

The reconfiguration process of software architectures for the distributed system is graphically shown in Figure 11.

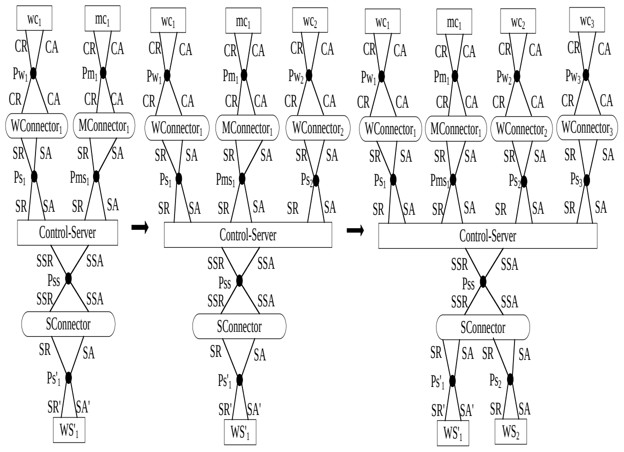

Figure 11. Dynamic Reconfiguration of System Architectures

When a client or a connector leaves, the situation is similar.

3) Hypergraph Grammar of Dynamic Software Architectures

According to the above analysis, we can define a hypergraph grammar of dynamic software architectures for the distributed system as $G = \{H, P, H_0\}$, where H is a set of hypergraphes of software architectures for the distributed system such as those hypergraphes mentioned as above, P is the set of all above-mentioned reconfiguration production rules, and $H_0 = \varnothing$ is the initial hypergraph of dynamic software architectures for those system.

V. Verification of Dynamic Software Architectures for Distributed Systems

During the dynamic reconfiguration of software architectures for distributed systems, a sequence of structural changes should satisfy some particular properties of interest. In our research, We consider mechanisms to express and verify some properties that we expect to be satisfied by dynamic reconfiguration of software architectures for distributed systems. In order to guarantee the correctness of the software architectures reconfiguration, we use model checking to verify these properties. Model checking[25] is a formal analysis technique for verifying finite state systems. In order to verify some property of a system, the main idea of model checking method is to represent for the system model with a state transition systems *M*, and to specify the property of the system with a logic formula *f*, then the verifying process is turn to verify whether the state transition system *M* satisfy the logic formula *f*, which can be wrote to *M* |=*f*. Here we write *M* |=*f* to denote that logic formula *f* holds in *M*.

We map hypergraphes of software architectures for a distributed system to states, and map reconfiguration production rules to transition relations, and define a state transition system,

Definition 5.1 (State Transition System of Software Architectures for a distributed system), a state transition system of software architectures for a distributed system is a tuple ***M=(S, s₀, R, AP, L)***, where

- ***S***={ $H_0, H_1, ..., H_t$ }, is a set of states, corresponding to a set of hypergraphes of software architectures for a distributed system,
- $s_0=H_0$ is the initial state, corresponding to the initial hypergraph of software architectures for the distributed system,
- $\boldsymbol{R \subseteq S \times S}$ is a set of transition relations, corresponding to reconfiguration production rules,
- **AP** is a set of atomic propositions of interesting,
- *L*: $\boldsymbol{S \rightarrow 2^{AP}}$ is a labeling function.

Given a hypergraph grammar of dynamic software architectures for a distributed system $G = (H, P, H_0)$, let $S=H$, $s_0=H_0$, $R=P$, and giving a set of atomic propositions *AP* of interesting, and labeling functions *L*, then we get the corresponding state transition system $M=(S, s_0, R, AP, L)$.

In our paper, we especially focus on the invariant property of dynamic software architectures for a distributed system. Its formal definition is,

Definition 5.2 (software architecture Invariant Property for a distributed system), a property of dynamic software architectures for a distributed system is an invariant property if there is a propositional logic formula ϕ corresponding to the property, for each finite evolution hypergraph sequence of dynamic software architectures for the distributed system H_0, H_1, …, H_n, H_i satisfy the formulation ϕ for each i, where H_0 is the initial hypergraph of dynamic software architectures, and $H_i \xrightarrow[p_i]{} H_{i+1}, 0 < i < n$, p_i is a reconfiguration production rule of dynamic software architectures for the distributed system. That is

$$\forall i, \quad H_i \models \phi, \quad 0 < i < n$$

For example, in our scenario in section 4, each web client should be connected to the connector *WConnector* through only one port in dynamic reconfiguration of software architectures for this system.

In order to ensure there is no loop in system communication, we should be sure that this property must be satisfied. We use a propositional logic formula to denote this property,

$$\begin{aligned}\Phi = (&\forall wc, wc \in H \rightarrow \exists Pw, Pw(wc, wconnetor) \in H \wedge \\ &(\forall Pw_1, Pw_2, Pw_1(wc, wconnector) \in H \wedge \\ &Pw_2(wc, wconnector) \in H \rightarrow Pw_1 = Pw_2))\end{aligned}$$

Where *wc*, *wconnector* present a web client, the connector *WConnector* respectively, Pw_i mean a port between the web client *wc* and the connector *wconnector*, and *H* is a hypergraph of dynamic software architectures for the distributed system.

In order to verify the property is an invariant property, we must check that Φ is hold at each reachable state from the initial state in the state transition system *M*. It is obvious that the initial state, i.e. the initial software architecture hypergraph satisfy the formulation Φ . We design a DFS (Depth-First Search) algorithm in *M* which can search each reachable software architecture hypergraph of *M*, and check validity of the property at the same time. When the algorithm is end, we can check whether the property is an invariant property.

Algorithm 1 summarizes the main steps for checking the invariant condition Φ by means of a depth-first search in the state graph *G(M)*. We start from the initial state and investigate all states that are reachable from it. If at least one state *s* is visited where Φ not hold, then the invariance induced by Φ does is violated. We define a Boolean variable *b* to monitor the algorithm running. When *b* is false, the algorithm will quit immediately. That means the invariance induced by Φ is violated, or the algorithm continue to traverse. In Algorithm 1, *R* stores all visited states, i.e., if Algorithm 1 terminates, then *R* = Reach(*M*) contains all reachable states. *U* is a stack that organizes all states that still have to be visited, provided they are not yet contained in *R*. The symbol ε is used to denote the empty stack. The operations *push*, *pop*, and *top* are the standard operations on stacks.

Algorithm 1: Invariant checking based on depth-first search

```
input: Finite states transition system M=(S, $s_0$, R, AP, L),
       initial state $s_0$, property $\Phi$ .
output: yes if M satisfies the invariant, otherwise no
Procedure IsInvariantDFS (transition system M,state $s_0$,
property $\Phi$ )
    R := $\emptyset$;
    U := ε;
    bool b := true;
    // perform a DFS for M
    push($s_0$,U);
    R := R ∪ { $s_0$ };
    do
        s1 := top(U);
        if post(s1) ⊆ R then
            pop(U);
            b := b ∧ (s1 |= $\Phi$ );
        else
            let s2 ∈ post(s1) \ R
            push(s2, U);
            R := R ∪ {s2};
        End if
    while ((U != ε ) ∧ b)
    if b then
        return("yes")
    else
        return("no")
    end if
end Procedure
```

VI. Conclusion and Future Work

Increasing complexity of distributed software systems makes it a growing concern at the dynamic evolution of distributed software systems, especially dynamic software architectures for those systems. In this paper, we described dynamic software architectures for distributed systems using a hypergraph grammar, and verified the invariant property of dynamic software architectures. Firstly, we gave out predefined reconfiguration production rules and operations for dynamic software architectures for distributed systems, and then we used an example to demonstrate dynamic reconfiguration process of software architectures for a distributed system using the hypergraph grammar. At last we verified the invariant property of dynamic software architectures for distributed systems using model checking. Our approach both has a graphical visual representation, and has a formal theoretical framework based on grammars.

The next step of our work is to describe dynamic software architectures for distributed systems with self-Repair dynamism reconfiguration production rules, and verify some other properties of dynamic software architectures for distributed systems such as consistency with model checking.

ACKNOWLEDGMENT

This work was supported by the National High-Tech Research and Development Plan of China (863 Program) under Grant No. 2007AA01Z425; the National Grand Fundamental Research Program of China (973 Program) under Grant No.2007CB316502; the National Natural Science Foundation Major Research Project of China under Grant No.90718015; the Joint of NSFC and Microsoft Asia Research under Grant No. 60970155; the Scientific Research Plan Projects of Education Department of Jiangxi province of China under Grant No. GJJ09263.

REFERENCES

[1] T. Mens, J. Buckley, M. Zenger, and A. Rashid. Towards a taxonomy of software evolution. Proceedings of the International Workshop on Unanticipated Software Evolution, 2003, pp. 309-326.
[2] D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architecture. ACM SIGSOFT Software Engineering Notes. 1992, 17(4):40-52.
[3] M. Shaw, D. Garlan. Software Architecture: Perspective on an Emerging Discipline. New York: Prentice Hall, 1996.
[4] D. Garlan. Software Architecture: A Roadmap. The Future of Software Engineering 2000, Proceedings of the 22nd International Conference on Software Engineering, New York, 2000, pp.91-101.
[5] YANG Qun, YANG Xianchun, XU Manwu.A Framework for Dynamic Software Architecture-based Self-healing. ACM SIGSOFT Software Engineering Notes. 2005, 30(4):1-4.
[6] J. Magee, J. Kramer. Dynamic structure in software architectures. Proceedings of the Fourth ACM SIGSOFT Symposium on Foundations of Software Engineering. 1996, pp. 3-14.
[7] R. Allen. A formal approach to software architectures [Ph.D. Thesis]. Pittsburgh: Carnegie Mellon University, 1997.
[8] R. Allen, R. Douence, D. Garlan. Specifying and analyzing dynamic software architectures. Lecture Notes in Computer Science. 1998. 1382:21-37.
[9] C. Canal, E. Pimentel, and J. M. Troya. Specification and refinement of dynamic software architectures. Proceedings of the TC2 First Working IFIP Conference on Software Architecture. 1999, pp. 107-126.
[10] F. Oquendo. π-ADL: an Architecture Description Language Based on the Higher-order Typedπ-Calculus for Specifying Dynamic and Mobile Software Architectures. ACM Sigsoft Software Engineering Notes, 2004, 29 (4): 1-14.
[11] P. Selonen, J. Xu. Validating UML models against architectural profiles. Proceedings of the 9th European Software Engineering Conference. 2003, pp. 58-67.
[12] M. H. Kacem, A. H. Kacem, M. Jmaiel, K. Drira. Describing dynamic software architectures using an extended UML model. Proceedings of Symposium on Applied Computing. 2006, pp.1245-1249.
[13] M. H. Kacem, M. N. Miladi, M. Jmaiel, A. H. Kacem, and K. Drira. Towards a uml profile for the description of software architecture. The Conference on Component-Oriented Enterprise Applications, 2005, pp. 25-39.
[14] D. L. Métayer. Describing software architecture styles using graph grammars. IEEE Transactions on Software Engineering. 1998, 24(7):521-533.
[15] M. Wermelinger, A. Lopes, J. L. Fiadeiro. A graph based architectural (Re)configuration language. ACM SIGSOFT Software Engineering Notes. 2001, 26(5):21-32.
[16] R. Bruni, A. Bucchiarone, S. Gnesi, H. Melgratti. Modelling Dynamic Software Architectures using Typed Graph Grammars. Electronic Notes in Theoretical Computer Science. 2008, 213(1):39-53.
[17] M. Endler. A Language for implementing generic dynamic reconfigurations of distributed programs. Proceedings of BSCN 94. 1994, pp.175-187.
[18] N. Aguirre, T. Maibaum. A Temporal Logic Approach to the Specification of Reconfigurable Component-Based Systems. Proceedings of the 17th IEEE international conference on Automated software engineering, 2002, pp. 271-278.
[19] C. Canal, E. Pimentel, J. M. Troya. Specification and Refinement of Dynamic Software Architectures. Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1). 1999, pp.107-126.
[20] Wang Yinghui, Zhang Shikun, Liu Yu ,et al . Ripple-effect analysis of software architecture evolution based reachability matrix . Journal of Software , 2004, 15(8) :1107-1115.
[21] Wang Yinghui, WANG Lifu. Research about model and ripple effect analysis of software architecture evolution. Acta Electronica Sinica. 2005, 33(8):1381-1386.
[22] T Batista, N Rodriguez. Dynamic Reconfiguration of Component-Based Applications. Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems. 2000. pp. 32-39.
[23] B. Schmerl and D. Garlan. Exploiting architectural design knowledge to support self-repairing systems. Proc. of SEKE'02, pp. 241–248. ACM, 2002.
[24] J. Bradbury, J. Cordy, J. Dingel, and M. Wermelinger. A survey of self-management in dynamic software architecture specifications. Proceeding of WOSS'04. ACM, 2004. pp. 28–33.
[25] E. Clarke, O. Grumberg, and D. Peled. Model Checking. MIT Press, 2000.

Hongzhen Xu, born on Oct. 1976, received his M.S. degree in computer application technology from East China Institute of Technology, China in 2003, and now is a Ph.D. candidate in computer software theory of Tongji University, China. He is an associate professor of department of computer Science and Technology of East China Institute of Technology. His main research interests include software architecture, software evolution and model checking.

Guosun Zeng, born on Sep. 1964, received his Ph.D. in computer software theory from Shanghai Jiao Tong University in 2000. He is a professor of department of computer Science and Technology of Tongji University. His main research interests include heterogeneity computing, information security and trusted software.

Bo Chen, born on Dec.1963, a Ph.D. candidate in computer software theory of Tongji University, China. He is an associate professor of department of computer Science and Technology of Guangxi Unoversity. His main research interests include web services, trusted software and model checking.