Specifying Dynamic Software Architectures for Distributed Systems

Hongzhen Xu1,2,3, Guosun Zeng1,3, and Bo Chen1,3

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, gszeng@tongji.edu.cn, cb31@sina.com

Abstract—Open and dynamic characteristics of distributed software systems make user requirements and hardware resources change more rapidly, which make them 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 action. 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 of distributed software systems is one possible solution to meet these demands. However, there are some challenges for building dynamically evolving distributed systems, while dynamic software architectures for them is one of the most crucial problems.

Dynamic software architectures 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. The typical reconfiguration operations for dynamic software architectures include adding, removing and 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 specifying dynamic software architectures for distributed systems. We represent software architectures for distributed systems as hypergraphes, and use hypergraph production rules to specify dynamic software architectures for distributed systems. This approach provides both a graphical representation and a formal basis that is in line with the usual way architectures are represented.

II. RELATED WORKS

Many research works are focusing on describing software architectures and their evolution. They can be sub-divided into three categories. The first uses ADLs (Architectural Description Languages) to model and analyze software architectures and their reconfiguration. For example, Darwin [2] is concerned with the structural aspects of software architectures, and lack description of how reconfigurations interact with on-going computations. Wright [3] is a static ADL, and make use of CSP (communication sequence process) to specify software architecture evolution. π-ADL [4] is 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 and its extended models to design software architectures and their reconfiguration. UML is expressive and easy to understand for its set of graphical notations, 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.

The third uses formal techniques such as graph based techniques [5], logic based [6] and algebra based [7]. D. L. Météayer [8] proposed to describe software architecture styles using graph grammars. M. Endler [9] aimed at providing specification languages for the software architecture and its evolution. C. Canal [10] described local evolution of software architectures without from global...
those methods based on logic and algebra cannot offer any graphical display to these models.

III. BACKGROUND

Definition 3.1 (Hypergraph) A hypergraph is a tuple $H = (V, E, s, t, l_V, l_E)$, where
- $V$ and $E$ are disjoint finite sets of nodes and hyperedges, respectively.
- $s, t : E \rightarrow V$ are two mappings indicating the source nodes and the target nodes of a hyperedge, where each hyperedge can be connected to a list of nodes.
- $l_V : V \rightarrow \Sigma_V$, $l_E : E \rightarrow \Sigma_E$ are two label functions of hyperedges and nodes which are mappings from $V$ and $E$ in two finite sets of labels.

In our paper, we represent 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; connector hyperedges as rounded boxes, which are connected to their attached nodes by thin lines which carry the attachment labels. Figure 1 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.

Formally, a graph is a set of relation tuples noted $R(e_1, ..., e_n)$ where $R$ is an $n$-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 1, 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 1 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 3.2 (Hypergraph Production Rule) A hypergraph production rule $p = (L, R)$, commonly written $p : L \rightarrow R$, 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 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 $p : L \rightarrow R$, 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 $L$ 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. In the following of this paper, we also call hypergraph production rules as reconfiguration production rules.

IV. SPECIFYING DYNAMIC SOFTWARE ARCHITECTURES FOR DISTRIBUTED SYSTEMS

A. Reconfiguration Production Rules and Operations

In order to model dynamic reconfiguration of software architectures for distributed systems, we predefine the following reconfiguration production rules and 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,

1) $H_1 \rightarrow H_1 \cup \{ (C(cr), R(c, cr), ..., P(c, cr), ...) \}$

2) $H_1 \rightarrow H_1 \cup \{ (NC(cr), R(c, cr), ..., P(c, cr), ...) \}$

When we add a new component or a new connector to a software architecture, we must add connections and
communication ports responding to the component or the connector. In production rules (1) and (2), \( C(c) \) means a component entity \( c \), \( NC(cr) \) means a component 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 2 shows the reconfiguration operation for adding a new component \( c1 \).

![Figure 2. Reconfiguration Operation for Adding a new component](image)

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,

\[
H_1 \rightarrow H_1 \cup \{C(c), R_i(c, \text{connector}),..., P_i(c, \text{connector})\} \quad (3) \\
H_1 \rightarrow H_1 \cup \{NC(cr), R_i(c, cr),..., P_i(c, cr)\} \quad (4)
\]

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 3 shows the reconfiguration operation for removing a component \( c2 \).

![Figure 3. Reconfiguration Operation for Removing a component](image)

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 the same signature, but with better performance. Replacing production rules for software architectures for distributed systems are following,

\[
H_1 \rightarrow H_1 \cup \{C(c), R_i(c, \text{connector}),..., P_i(c, \text{connector})\} \\
- \{C(c), R_i(c, \text{connector}),..., P_i(c, \text{connector})\} \\
H_1 \rightarrow H_1 \cup \{NC(cr), R_i(c, cr),..., P_i(c, cr)\} \\
- \{NC(cr), R_i(c, cr),..., P_i(c, cr)\}
\]

For example, Figure 4 shows the reconfiguration operation for replacing a connector with \( connector1 \).

![Figure 4. Reconfiguration Operation for Replacing a connector](image)

2) Modeling Dynamic Software Architectures

Supposing that there is no entity activated before the system initialization. After initialization, the system only activates the Web-Server. Let \( H_0 = \emptyset \), 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),

\[
H_0 = \emptyset \rightarrow \{S(ws)\} = H_i
\]

Where we use \( ws \) to represent for the Web-Server. We get the following hypergraph shown in Figure 6.

![Figure 6. Initial Configuration of the System](image)
corresponding connector, connections and communication ports. The reconfiguration process is formally following,

\[ H_1 \rightarrow H_1 \cup \{NC(WConnector), SR(WConnector, ws), SA(ws, WConnector, ws), Ps(WConnector, ws)\} = H_2 \]

\[ H_2 \rightarrow H_2 \cup \{C(wc), CR(wc, WConnector), CA(WConnector, wc)\} \]

\[ Pm_1(wc, WConnector) = H_3 \]

\[ H_4 \rightarrow H_4 \cup \{NC(MConnector), SR(MConnector, ws), SA(ws, MConnector, Pms)\} = H_5 \]

\[ Pm_2(ws, MConnector) = H_6 \]

The reconfiguration process of software architectures for the distributed system is graphically shown in Figure 7.

When the Web-Server is down, the system will automatically replace Web-Server with Web-Server’ by application to the reconfiguration production rule (5). The reconfiguration process of software architectures is formally following,

\[ H_3 \rightarrow H_3 \cup \{S(ws), SR(WConnector, ws), SA(ws, WConnector)\} \]

\[ SR(MConnector, ws), SA(ws, MConnector)\]

\[ Ps(ws, WConnector, Pms) = H_5 \]

The reconfiguration process of software architectures is graphically shown in Figure 8.

V. ANALYSIS 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.

We map hypergraphs of software architectures for a distributed system to states, and map reconfiguration production rules to transition relations, and get the corresponding state transition system which can be used to verify some properties of dynamic software architectures for a distributed system. The next step of our work is to verify some properties of dynamic software architectures for distributed systems such as consistency with model checking.

VI. CONCLUSION

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 hypergraph and hypergraph production rules. We represent software architectures for a distributed system as hypergraphs, and use reconfiguration production rules and operations to model dynamic software architectures for distributed systems, and then we used an example to demonstrate our approach. Our approach both has a graphical visual representation, and has a formal theoretical framework.

REFERENCES


