TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214 08/10 pp599–609
DOI: 10.26599/TST.2018.9010027
Volume 23, Number 5, October 2018

Software System Evolution Analysis Method Based on Algebraic Topology

Chun Shan, Liyuan Liu, Jingfeng Xue*, Changzhen Hu, and Hongjin Zhu

Abstract: The analysis of software system evolution is highly significant in software research as the evolution runs throughout the lifecycle of a software system. Considering a software system as an algebraic engineering system, we propose a software system evolution analysis method based on algebraic topology. First, from a complex network perspective, we abstract a software system into the software structural topology diagram. Then, based on the algebraic topology principle, we abstract each node in the software structural topology diagram into an algebraic component represented by a 6-tuple. We propose three kinds of operation relationships between two algebraic components, so that the software system can be abstracted into an algebraic expression of components. In addition, we propose three forms of software system evolution, which help to analyze the structure and evolution of system software and facilitate its maintenance and reconfiguration.

Key words: software structural topology diagram; algebraic component; topological complex; evolution

1 Introduction

Static evolution and dynamic evolution have been proposed in previous studies[1,2] as the two main types[3] of software evolution. Static evolution refers to the evolution under non-operating conditions. In this case, the migration of the operating states of the software system as well as the active processes could be ignored, but the function of the software system may be temporarily disabled and therefore, not sustainable. Dynamic evolution refers to the evolution under the operating conditions[4]. It overcomes the shortcomings of static evolution, but dynamic changes such as the migration of the operating states and the active process[5] need to be considered, which can result in technical difficulties.

In this paper, we propose a dynamic software system evolution analysis method which considers a software system as an algebraic engineering system. First, from a complex network perspective, the software system is abstracted into a structural topology diagram with the information resources as the node and the behavior as the edge. Then, using a complex based on the topological properties, each node in the diagram corresponding to the software system module is abstracted into an algebraic component PM, and the algebraic component is represented by a 6-tuple: $\mathrm{PM} = (X, Y, D, C, \tau, \eta)$. The integrated software system is considered to be assembled from these algebraic components. Then, we proposed three operation relationships: calling, nesting, and including. These operation relationships can form a complete algebraic system by which an arbitrary software system can be abstracted into the form of algebraic expressions of the components. Considering the algebraic components, the analysis of software system evolution is therefore transformed into three forms: addition of components, deletion of components, and modification of components. We therefore establish the theoretical framework of using algebraic components to describe the software and analyze the software

• Chun Shan, Liyuan Liu, Jingfeng Xue, Changzhen Hu, and Hongjin Zhu are with the School of Software, Beijing Institute of Technology, Beijing 100081, China. E-mail: sherryshan@bit.edu.cn; 597098022@qq.com; xuejf@bit.edu.cn; chzhoo@bit.edu.cn; 1961357332@qq.com.
* To whom correspondence should be addressed.
Manuscript received: 2017-11-03; accepted: 2017-11-14

evolution process.

2 Related Work

Various foreign software component models have been proposed in the literature, among which, the most popular three are the Common Object Request Broker Architecture[6] proposed by Object Management Group, Distributed Component Object Model[7] proposed by Microsoft, and the Enterprise JavaBeans model[8] proposed by Sun Microsystems. These three models separate the software components interface from its implementation, and the components interact only through the interface; therefore, the reusability of the components is improved.

Furthermore, many researchers in China have extensively studied software component models, and have achieved noteworthy results. The most widely accepted is the "Bluebird component model" proposed by Yang et al.[9] The model combines the advantages of the three abovementioned foreign models and divides the components into two parts: external interface and internal structure. Based on this model, Zhang and Li[10] proposed an algebraic representation of software components, dividing the software components into functional parts and connecting parts. However, this expression separating the functional and connecting parts of software system is only applicable to an object-oriented programming language and not a process-oriented programming language.

3 Software System Evolution Analysis Method Based on Algebraic Topology

3.1 Method overview

The method is a kind of method for analyzing dynamic software system evolution[11, 12] based on algebraic topology, which regards a software system as an algebraic engineering system for implementing logical organization. It abstracts software system into an algebraic expression to analyze the evolution.

The method involves three specific steps. First, the software system is abstracted into a structural topology diagram based on the idea of complex networks. Second, the software system is presented with an algebraic expression of components based on the structural topology diagram. Third, the evolution of software system is mapped to the changes of the algebraic expression of components, and the software system evolution is analyzed from three perspectives: addition of component, deletion of component, and modification of component.

3.2 Algebraic abstraction

Researchers have found that the topologies of software systems usually have complex network-related properties[13] such as "small-world"[14] and "scale-free"[15], which inspires us to combine ideas on software engineering[16] and complex network[17] to analyze software systems.

Our research on software systems is therefore centered on structural topology by abstracting a structural topology diagram from the software system. In a software system, we abstracted various modules (subsystems) into nodes, and the interaction between the modules into edges between nodes. Then, the static structure of the whole software system can be abstracted into a network structure formed by nodes and edges, which becomes the software system structural topology diagram. For a software system, both the number of modules and the interaction between various modules are fixed; therefore, there is only one corresponding specific single software structural topology diagram.

Currently, complex network theory[18, 19] and research methods are mainly used in open-source software systems. For software systems developed by a process-oriented programming language, we can abstract the functions (methods) and structures in source codes into nodes. The relationship between the use of different methods and structures as well as the interaction between the methods and structures can be abstracted into edges (Fig. 1). For software systems developed by an object-oriented programming language, we can abstract the classes into nodes, and the association, generalization, inheritance, dependency, aggregation, and composite relationships between different classes into edges between nodes (Fig. 2). This way, any software system can be abstracted into a software structural topology diagram[20] based on the source code.

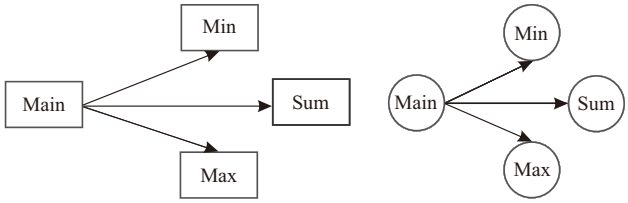

Fig. 1 Process-oriented procedure calling graph and corresponding topology diagram.

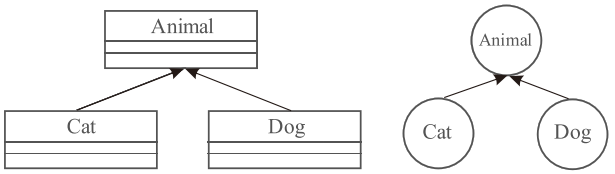

Fig. 2 Object-oriented class diagram and corresponding topology diagram.

3.3 Algebraic method of software system

Algebraic topology is a branch of mathematics that uses abstract algebra tools to study topological spaces, where the algebra tools convert topological problems into algebraic ones, and the topological problems are solved with algebraic methods.

In this paper, we regard the software system topology diagram as a complex based on topological properties, which has been used for research in biological fields[21], and the software components (such as modules, subsystems) as a subcomplex. Each complex can be abstracted into an algebraic component based on the algebraic topology principle. The integration of software systems is composed of these algebraic components. To describe algebraic components, we propose a software system composite framework (Fig. 3). The behavior of each level of the software system can be presented as the structure of the algebraic components. The whole software system is an integral structure composed of intertwining vertical and horizontal algebraic components, in which the upper layer of the algebraic components is a composite of those in the lower layer.

Then, we propose an algebraic aggregation model based on the software system composite framework (Fig. 4). The algebraic component of the software system is considered to be a 6-tuple: $\mathrm{PM} = (X, Y, D, C, \tau, \eta)$, which consists of functional and connection parts. Here, $X = \{x_1, x_2, \ldots, x_n\}$ is a

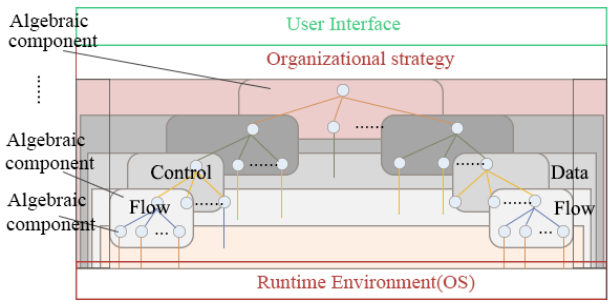

Fig. 3 Software system composite framework.

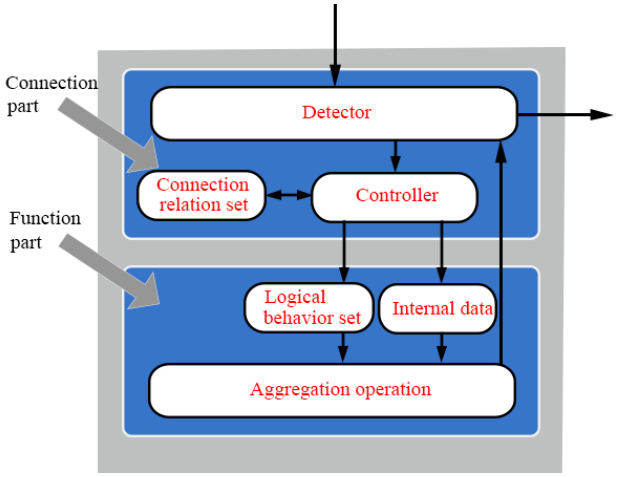

Fig. 4 Software system algebraic aggregation model.

logical behavior set composed of logical behavior $x_1, x_2, \ldots, x_n$. In addition, Y is a set of connection relations, C denotes the controller, D denotes the internal data, τ represents the aggregation operation, and η denotes the detector. The functional parts of the software system algebraic component constitute the logical behavior set, internal data, the aggregation operation, the connection parts by the connection relation set, the detector, and the controller.

According to the relationship between the modules in the software system, the operation relationships between algebraic components can be divided into three: calling operation, including operation, and nesting operation.

Calling operation: Assuming there are two different algebraic components PM_1 and PM_2 in the software system S, if PM_1 uses the services provided by PM_2, then a calling relationship exists between PM_1 and PM_2, recorded as $\mathrm{PM}_1 \bigoplus \mathrm{PM}_2$, with the corresponding topological form shown in Fig. 5.

Nesting operation: Assuming there are two different algebraic components PM_1 and PM_2 in the software system S, if the algebraic components PM_1 and PM_2 are nested together to accomplish one function, then

Fig. 5 Calling relationship topological form.

a nesting relationship exists between PM_1 and PM_2, denoted as $\mathrm{PM}_1 \bigotimes \mathrm{PM}_2$. The corresponding software structure topological form is shown in Fig. 6.

Including operation: Assuming there are two different algebraic components PM_1 and PM_2 in the software system S, if the execution of PM_1 is based on the execution of PM_2; that is, the output of PM_2 is the input of PM_1, then PM_1 is contained in PM_2; it could be concluded an including relationship exists between PM_1 and PM_2, denoted as $\mathrm{PM}_1 \bigodot \mathrm{PM}_2$. The corresponding topological form is shown in Fig. 7.

According to the above three relationships and their corresponding topology diagrams, any software system can be abstracted into the algebraic expressions. The operation rule is as follows: The priority of computing between algebraic components is left-right, including-nesting-calling, and parentheses have the highest priority if they exist.

Three steps are needed to abstract an algebraic expression from a given software system structural topology diagram. (1) Roughly decompose the diagram from top to bottom: divide the structural topology diagram (subgraph) into upper and lower parts until the leaf nodes. (2) Refine the new diagram from left to right: decompose the lower part of the topological graph from left to right until there is no topology subgraph on the right side. (3) Integrate the system: replace each segmented part of the top algebraic expression with the algebraic expressions corresponding with the topological subgraphs, so that the integrated algebraic expression for the software system is obtained.

We need to specify the condition that algebraic component PM_1 equals PM_2 before studying the computational properties of algebraic components:

(1) $\mathrm{PM}_1.X = \mathrm{PM}_2.X$;

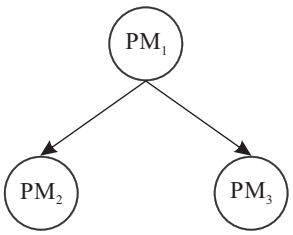

Fig. 6 Nesting relationship topological form.

Fig. 7 Including relationship topological form.

(2) $\mathrm{PM}_1.Y = \mathrm{PM}_2.Y$;

(3) $\mathrm{PM}_1.D = \mathrm{PM}_2.D$;

(4) $\mathrm{PM}_1.C = \mathrm{PM}_2.C$;

(5) $\mathrm{PM}_1.\tau = \mathrm{PM}_2.\tau$;

(6) $\mathrm{PM}_1.\eta = \mathrm{PM}_2.\eta$.

By induction and proving, we obtain the following 10 operation properties on the three operation relationships:

Property 1: Assuming that there are three different algebraic components $\mathrm{PM}_1, \mathrm{PM}_2$, and PM_3 in the software system S, then PM_1, PM_2, and PM_3 satisfy the union law on the calling operation.

$$(\mathrm{PM}_1 \oplus \mathrm{PM}_2) \oplus \mathrm{PM}_3 = \mathrm{PM}_1 \oplus (\mathrm{PM}_2 \oplus \mathrm{PM}_3) \quad (1)$$

Proof 1: To prove Property 1, we only need to prove that every tuple of the algebraic components on one side of the equality sign is equal to the corresponding tuple on the other side; that is, the two algebraic components satisfy the equality condition; then, the above six equations are true. First, for $\forall x \in ((\mathrm{PM}_1 \oplus \mathrm{PM}_2) \oplus \mathrm{PM}_3).X$, we get $x \in (\mathrm{PM}_1 \oplus \mathrm{PM}_2).X$ or $x \in \mathrm{PM}_3.X$. If $x \in \mathrm{PM}_3.X$, then $x \in (\mathrm{PM}_2 \oplus \mathrm{PM}_3).X$, so $x \in \mathrm{PM}_1 \oplus (\mathrm{PM}_2 \oplus \mathrm{PM}_3).X$. If $x \in (\mathrm{PM}_1 \oplus \mathrm{PM}_2).X$, we get $x \in \mathrm{PM}_1.X$ or $x \in \mathrm{PM}_2.X$. Similarly, if $x \in \mathrm{PM}_1.X$, we can obtain $x \in \mathrm{PM}_1 \oplus (\mathrm{PM}_2 \oplus \mathrm{PM}_3).X$; if $x \in \mathrm{PM}_2.X$, then $x \in (\mathrm{PM}_2 \oplus \mathrm{PM}_3).X$, we can also obtain $x \in \mathrm{PM}_1 \oplus (\mathrm{PM}_2 \oplus \mathrm{PM}_3)$.

Property 2: Assuming that there are three different algebraic components $\mathrm{PM}_1, \mathrm{PM}_2$, and PM_3 in the software system S, then $\mathrm{PM}_1, \mathrm{PM}_2$, and PM_3 satisfy the union law on the nesting operation.

$$(\mathrm{PM}_1 \otimes \mathrm{PM}_2) \otimes \mathrm{PM}_3 = \mathrm{PM}_1 \otimes (\mathrm{PM}_2 \otimes \mathrm{PM}_3) \quad (2)$$

Proof 2: The proving process is similar to that of Proof 1.

Property 3: Assuming that there are two different algebraic components PM_1 and PM_2 in the software system S, then PM_1 and PM_2 satisfy the commutative law on the nesting operation.

$$\mathrm{PM}_1 \otimes \mathrm{PM}_2 = \mathrm{PM}_2 \otimes \mathrm{PM}_1 \quad (3)$$

Proof 3: The proving process is similar to Proof 1.

Property 4: Assuming that there are three different algebraic components PM_1, PM_2, and PM_3 in the software system S, then $\mathrm{PM}_1, \mathrm{PM}_2$, and PM_3 satisfy the union law on the including operation.

$$(\mathrm{PM}_1 \odot \mathrm{PM}_2) \odot \mathrm{PM}_3 = \mathrm{PM}_1 \odot (\mathrm{PM}_2 \odot \mathrm{PM}_3) \quad (4)$$

Property 5: Assuming that there are three different algebraic components $\mathrm{PM}_1, \mathrm{PM}_2$, and PM_3 in the software system S, for $\mathrm{PM}_1, \mathrm{PM}_2$, and PM_3, the nesting

operations of the three algebraic components satisfy the union law to the including operation.

$$(\mathrm{PM}_1 \otimes \mathrm{PM}_2) \odot \mathrm{PM}_3 = (\mathrm{PM}_1 \odot \mathrm{PM}_3) \otimes (\mathrm{PM}_2 \odot \mathrm{PM}_3) \tag{5}$$

Property 6: Assuming that there are three different algebraic components $\mathrm{PM}_1, \mathrm{PM}_2$, and PM_3 in the software system S, for $\mathrm{PM}_1, \mathrm{PM}_2$, and PM_3, the nesting operations of the three algebraic components satisfy the union law to the calling operation.

$$(\mathrm{PM}_1 \otimes \mathrm{PM}_2) \oplus \mathrm{PM}_3 = (\mathrm{PM}_1 \oplus \mathrm{PM}_3) \otimes (\mathrm{PM}_2 \oplus \mathrm{PM}_3) \tag{6}$$

Property 7: Assuming that there are two different algebraic components PM_1 and PM_2 in the software system S, for PM_1 and PM_2, the results of calling operation, nesting operation, and including operations of two algebraic components are also algebraic components; that is, $\mathrm{PM}_1 \oplus \mathrm{PM}_2$, $\mathrm{PM}_1 \otimes \mathrm{PM}_2$, and $\mathrm{PM}_1 \odot \mathrm{PM}_2$, are also algebraic components and belong to the software system S.

Property 8: Assuming that there are n different algebraic components $\mathrm{PM}_1, \mathrm{PM}_2, \ldots, \mathrm{PM}_n$ in the software system S, for $\mathrm{PM}_1, \mathrm{PM}_2, \ldots, \mathrm{PM}_n$, the results of calling operation, nesting operation, and including operations of two algebraic components are also algebraic components; that is, $\mathrm{PM}_1 \oplus \mathrm{PM}_2 \oplus \cdots \oplus \mathrm{PM}_n$, $\mathrm{PM}_1 \otimes \mathrm{PM}_2 \otimes \cdots \otimes \mathrm{PM}_n$, and $\mathrm{PM}_1 \odot \mathrm{PM}_2 \odot \cdots \odot \mathrm{PM}_n$ are also algebraic components and belong to the software system S.

Property 9: The calling, nesting, and including operation relationships of the algebraic components in the software system S constitute a complete algebraic system.

Property 10: The set of all algebraic components of the software system S and the three computing relationships between algebraic components, i.e., the calling, nesting, and including computing relationships, constitute an exchange semi-group denoted as $< S, \oplus >$, $< S, \otimes >$, and $< S, \odot >$, respectively.

The algebraic system composed of algebraic components and the calling, nesting, and including relationships must be a complete algebraic system since the software system is a complete system. Only when this precondition is satisfied can we abstract any software system into the algebraic expression with our method.

Here we show the proving process after clarifying the following two definitions.

(1) Software system structure set: Assuming that the set of all algebraic components in the software system S is defined as $\mathrm{PM} = \{\mathrm{PM}_1, \mathrm{PM}_2, \ldots, \mathrm{PM}_n\}$, and the set of arithmetic relationships between algebraic components is defined as $R = \{\oplus, \otimes, \odot\}$. If two algebraic components PM_i and PM_j in the software system S are related by R, denoted as $< \mathrm{PM}_i, \mathrm{PM}_j >$, then $< \mathrm{PM}_i, \mathrm{PM}_j > \in R$, the system generated by the algebraic components set PM on the arithmetic relationship set R, is called the software system structure set, denoted as $S_c = R(C)$.

(2) The integratedness of the computing system: If the various algebraic components that make up the software system satisfy the three conditions below, then the computing system is integrated. First, the algebraic components in the software system S are closed to any of the operations; second, the software system S generated by the three arithmetic relationships between algebraic components is closed; third, every algebraic component PM_i in the software system S has a specific function; that is, the algebraic component PM_i is complete.

The proving process can be divided into three steps in detail.

(1) To prove that all algebraic components in the software system S are closed to the calling, nesting, and including computing relations, i.e., the algebraic components generated by these computing relationships belong to this software system, we assume, based on Property 7, that there are two different algebraic components PM_1 and PM_2 in the software system S; then, the algebraic components $\mathrm{PM}_1 \oplus \mathrm{PM}_2$, $\mathrm{PM}_1 \otimes \mathrm{PM}_2$, and $\mathrm{PM}_1 \odot \mathrm{PM}_2$ also belong to S. Then, we can state that the algebraic components in the software system S are closed to each computing relationship.

(2) To prove the software system generated by nesting, calling, and including computing relationships is closed, we assume that $O = (O_1, O_2, \ldots, O_i)$ is the algebraic components' set composed by i ($i > 1$, $i \in \mathbb{N}$) algebraic components on $\oplus$ relation, $P = (P_1, P_2, \ldots, P_j)$ is the algebraic components' set composed by j ($j > 1$, $j \in \mathbb{N}$) algebraic components on $\otimes$ relation, and $Q = (Q_1, Q_2, \ldots, Q_k)$ is the algebraic components' set composed by k ($k > 1, k \in \mathbb{N}$) algebraic components on $\odot$ relation, apparently, O, P, $Q \subseteq S_c$, and for any algebraic component O_i, P_j, and Q_k, there are $O_i \in S_c$, $P_j \in S_c$, and $Q_k \in S_c$. Based on Property 7, we know that the algebraic components generated by the three computing relationships belong to S_c; then, we can state that the software system generated by the three computing relationships is

closed.

(3) To prove that every algebraic component in the software system S has a specific function, the algebraic component $\mathrm{PM} = (X, Y, D, C, \tau, \eta)$ in the software system is derived from the software system algebraic aggregation model proposed in Fig. 4; therefore, any algebraic component can represent a functional module and has a specific function.

3.4 Three forms of evolution

When a software system is being utilized, various internal and external factors will lead to changes in users' needs, and the software system needs to evolve to satisfy users' needs. In this study, we investigate software system evolution based on the algebraic expression of components, and the evolution is attributed to the change of the relationships among the various algebraic components that make up the software system.

Three forms of software system evolution are addition, deletion, and modification of algebraic components.

(1) Addition of software algebraic components: it is divided into the addition of calling algebraic component, addition of nesting algebraic component, and addition of including algebraic components, since the relationships among the algebraic components are divided into calling, nesting, and including. The evolution process of addition of calling algebraic component is shown here.

Suppose the software system structure topology diagram is as that shown in Fig. 8.

In the software system shown in Fig. 8, a new calling component PM_{11} is added, and it is called by component PM_5 in the original software system; therefore, the new software system topology can be obtained as Fig. 9.

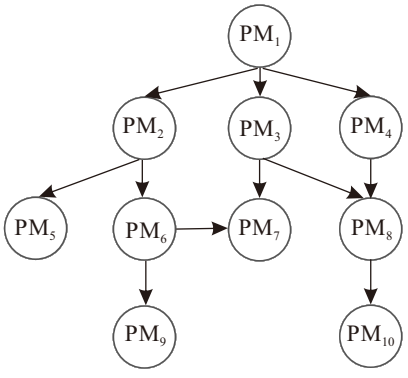

Fig. 8 A software system structure topology diagram.

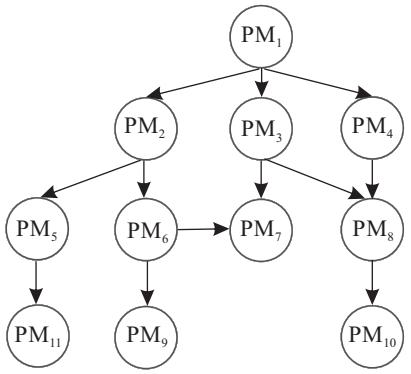

Fig. 9 The structure topology diagram after adding component PM_{11}.

The algebraic expression of components of the original software system is $\mathrm{PM}_1 \oplus [\mathrm{PM}_2 \oplus \mathrm{PM}_5 \otimes (\mathrm{PM}_6 \oplus \mathrm{PM}_9)] \otimes [\mathrm{PM}_3 \oplus (\mathrm{PM}_7 \odot \mathrm{PM}_6) \otimes (\mathrm{PM}_8 \oplus \mathrm{PM}_{10})]$. After evolution, the expression is $\mathrm{PM}_1 \oplus [\mathrm{PM}_2 \oplus (\mathrm{PM}_5 \oplus \mathrm{PM}_{11}) \otimes (\mathrm{PM}_6 \oplus \mathrm{PM}_9)] \otimes [\mathrm{PM}_3 \oplus (\mathrm{PM}_7 \odot \mathrm{PM}_6) \otimes (\mathrm{PM}_8 \oplus \mathrm{PM}_{10})] \otimes [\mathrm{PM}_4 \oplus (\mathrm{PM}_8 \oplus \mathrm{PM}_{10})]$.

(2) To describe the full-scale evolution of the software system, we divide the deletion of algebraic component into five parts: deletion of the top algebraic component, deletion of the underlying calling algebraic component, deletion of the underlying nesting algebraic component, deletion of the underlying including algebraic component, and the associated deletion. Here, we mainly focus on deletion of the top algebraic component and the associated deleting.

The function of the top algebraic component of the software system is to combine all the algebraic components to form a complete system as the entrance of the software system. Therefore, removing the top component results in loss of contact to the other algebraic components, which collapses the entire software system. Thus, we need to introduce a new algebraic component to serve as the entrance of the software system after deleting the top algebraic component.

For example, considering the software system topology shown in Fig. 10, after deleting the top algebraic component PM_1, we need to introduce a new component PM_0 to be the new top component, so that we can get the new topology given in Fig. 11.

The algebraic expression of components of the original software system is $\mathrm{PM}_1 \oplus [\mathrm{PM}_2 \oplus (\mathrm{PM}_5 \oplus \mathrm{PM}_{11}) \otimes [\mathrm{PM}_6 \oplus (\mathrm{PM}_9 \odot \mathrm{PM}_{13})]] \otimes [\mathrm{PM}_3 \oplus (\mathrm{PM}_7 \odot \mathrm{PM}_6) \otimes (\mathrm{PM}_8 \oplus \mathrm{PM}_{10})] \otimes [\mathrm{PM}_4 \oplus (\mathrm{PM}_8 \oplus \mathrm{PM}_{10}) \otimes \mathrm{PM}_{12}]$. After evolution, the expression is $\mathrm{PM}_0 \oplus [\mathrm{PM}_2 \oplus (\mathrm{PM}_5 \oplus \mathrm{PM}_{11}) \otimes [\mathrm{PM}_6 \oplus (\mathrm{PM}_9 \odot \mathrm{PM}_{13})]] \otimes [\mathrm{PM}_3 \oplus (\mathrm{PM}_7 \odot \mathrm{PM}_6) \otimes (\mathrm{PM}_8 \oplus \mathrm{PM}_{10})] \otimes [\mathrm{PM}_4 \oplus (\mathrm{PM}_8 \oplus$

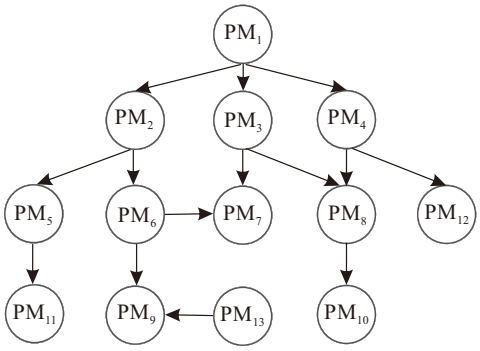

Fig. 10 The structure topology diagram after adding the algebraic components.

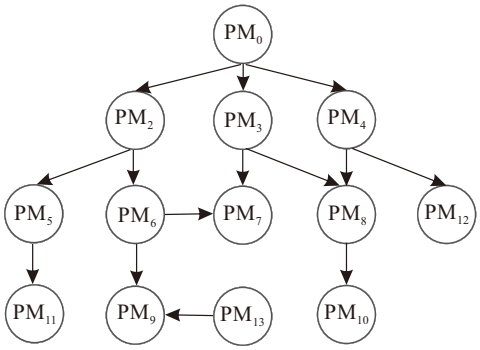

Fig. 11 The structure topology diagram after deleting the component PM_1.

$PM_{10}) \otimes PM_{12}]$.

The actual software system evolution process involves the deletion of intermediate algebraic components, but the intermediate algebraic components may be related to many algebraic components. Therefore, we need to consider the associated deletion. When deleting an intermediate algebraic component, we need to delete all of its relations.

For example, the structural topology diagram of a software system is shown in Fig. 12, after deleting the intermediate algebraic component PM_6, we also need to delete the associated relations, and we can get the new structural topology diagram in Fig. 13.

The algebraic expression of components of the original software system is $PM_0 \oplus [PM_2 \oplus (PM_5 \oplus PM_{11}) \otimes (PM_6 \oplus PM_9)] \otimes (PM_3 \oplus PM_8) \otimes (PM_4 \oplus PM_8 \otimes PM_{12})$. After evolution, the expression is $PM_0 \oplus [PM_2 \oplus (PM_5 \oplus PM_{11})] \otimes (PM_3 \oplus PM_8) \otimes (PM_4 \oplus PM_8 \otimes PM_{12})$.

(3) When modifying algebraic components, the algebraic expression of components does not change if only the internal structure, and not the external relations, is modified. However, if we modify the relationships between algebraic components, the

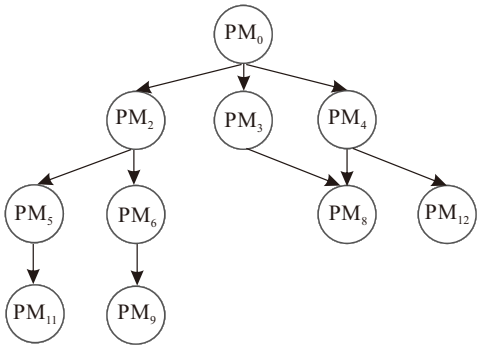

Fig. 12 The structure topology diagram after deleting the algebraic components.

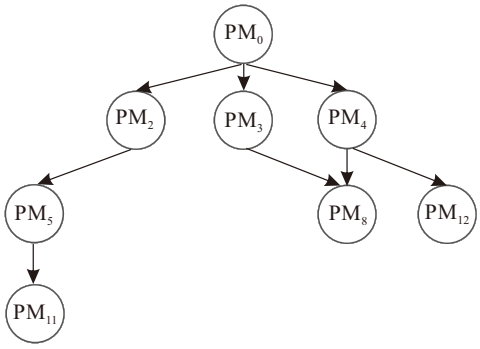

Fig. 13 The structure topology diagram after deleting the component PM_6.

expression would change correspondingly. Research on this kind of modification can be conducted from two perspectives: modification of the top algebraic component relationships and that of the non-top algebraic component relationships.

In the topology shown in Fig. 14, the top algebraic component PM_0 is modified to eliminate the calling relationship between PM_0 and PM_4. To ensure that the software system runs smoothly, we need to introduce a new algebraic component PM as the software system entrance to integrate the entire software system, as shown in Fig. 15.

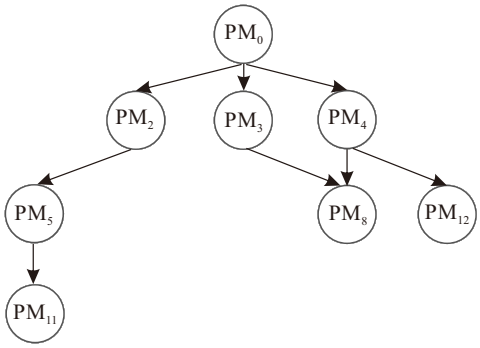

Fig. 14 Software system structure topology diagram after deleting the associated algebraic components.

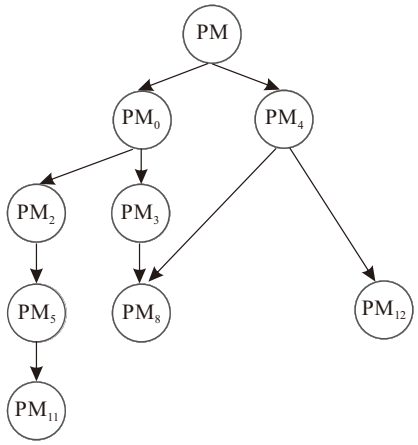

Fig. 15 Structure topology diagram after modifying the component PM_0.

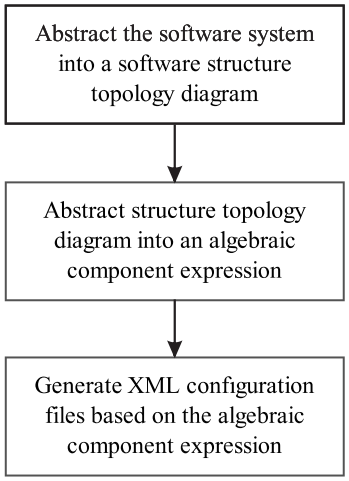

Fig. 16 Experiment procedure.

The algebraic expression of components of the original software system is $PM_0 \oplus [PM_2 \oplus (PM_5 \oplus PM_{11})] \otimes (PM_3 \oplus PM_8) \otimes (PM_4 \oplus PM_8 \otimes PM_{12})$. After evolution, the expression is $PM \oplus [PM_0 \oplus [PM_2 \oplus (PM_5 \oplus PM_{11})] \otimes (PM_3 \oplus PM_8)] \otimes (PM_4 \oplus PM_8 \otimes PM_{12})$.

There is no peculiarity for non-top algebraic components in a software system, and we only need to establish new communications according to new relationships between the algebraic components.

By abstracting the software system into an algebraic expression of components, the research on evolution of the software system becomes centered on the algebraic expression of components. The evolution law of software system can be obtained when algebraic theory is applied to analyze the algebraic expression of components; this reduces the risk involved with the direct analysis of software system evolution as well as the analysis time.

4 Experiment and Analysis

4.1 Experiment environment

This experiment was carried out in the Windows 10 operating system with JavaEE development environment, using an open-source software system JeeSite, which is a rapid development platform based on JavaEE as well as several excellent open-source projects. It is highly efficient and secure.

4.2 Experiment process, results, and analysis

The experiment procedure comprises three steps, as shown in Fig. 16.

First, the software system for experiment was selected, and its structural topology diagram was abstracted out of the software system. Then, algebraic components were abstracted from every node in the topology diagram, and the algebraic expression of components was obtained. The XML configuration file was generated according to the algebraic expression of components of the software system. When the software system evolution was needed, we modified its algebraic expression of components and according to the modified expression, we generated a new XML configuration file. Then, the software system reorganized its functions according to the newly generated XML configuration file. In this way, we realized a dynamic evolution of the software system.

The first step was to obtain the topology diagram of software structure. The four modules of JeeSite, system management (SYS), content management (CMS), online office (OA), and code generation (GEN), are shown in Fig. 17. The system management module includes enterprise organization structure (CG), menu management (MG), and role rights management (RG) function. The content management module includes content management (NG), column management (LG),

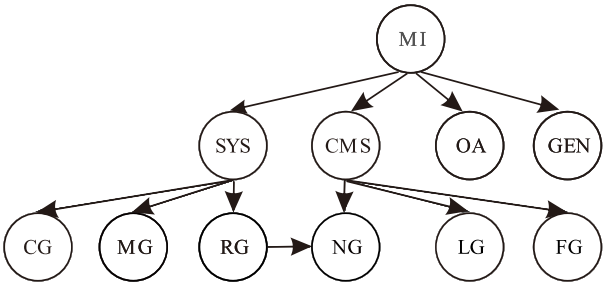

Fig. 17 JeeSite structure topology diagram.

and document management function (FG). We obtained the topology diagram of software structure as shown in Fig. 17 according to the relationship between various modules, with MI representing the main interface module.

The second step was to abstract the algebraic expression of components from the topology diagram of the software structure. First we roughly decomposed the topology diagram of software structure from top to bottom to obtain the expression MI $\oplus$ SYS $\otimes$ CMS $\otimes$ OA $\otimes$ GEN. Then, we refined SYS and CMS modules stepwise from left to right to get expressions SYS $\oplus$ CG $\otimes$ MG $\otimes$ RG and CMS $\oplus$ (NG $\odot$ RG) $\otimes$ LG $\otimes$ FG. Lastly, we integrated the entire algebraic expression upward to obtain the algebraic expression of components of JeeSite as MI $\oplus$ (SYS $\oplus$ CG $\otimes$ MG $\otimes$ RG) $\otimes$ [CMS $\oplus$ (NG $\odot$ RG) $\otimes$ LG $\otimes$ FG] $\otimes$ OA $\otimes$ GEN.

The third step was to form the XML configuration file according to the algebraic expression. The structure of XML configuration file is shown in Fig. 18, where "component operation list" represents the three operation relationships between components: calling, nesting, and including; "configuration" represents the configuration relationship among the components; and "component relation list" represents the relationship list between the components. According to the algebraic expression of components obtained in the second step, we obtained the XML configuration file as shown in Fig. 19.

During operation, the algebraic expression of components of the software evolution changed, which consequently changed the XML configuration file of software system to complete the software system evolution. Therefore, the software system evolution is mapped to the changes of the algebraic expression of components, and the software system can evolve

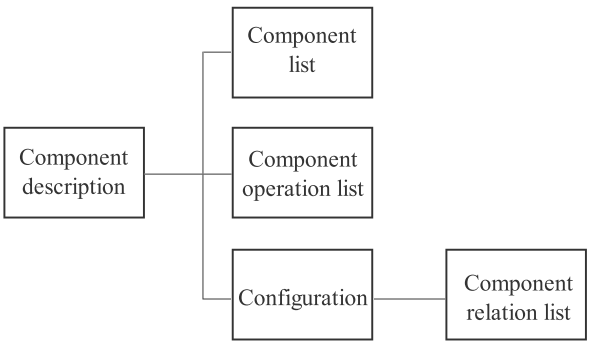

Fig. 18 Structure of XML configuration file.

```
<ComponentDescription>
    <ComponentList>
        <ComponentName>MI</ComponentName>
        <ComponentName>SYS</ComponentName>
        <ComponentName>CMS</ComponentName>
        <ComponentName>OA</ComponentName>
        <ComponentName>GEN</ComponentName>
        <ComponentName>CG</ComponentName>
        <ComponentName>MG</ComponentName>
        <ComponentName>RG</ComponentName>
        <ComponentName>NG</ComponentName>
        <ComponentName>LG</ComponentName>
        <ComponentName>FG</ComponentName>
    </ComponentList>

    <ComponentOperationList>
        <ComOperation Attr="Call">Call Operation</ComOperation>
        <ComOperation Attr="Nest">Nest Operation</ComOperation>
        <ComOperation Attr="Contain">Contain Operation</ComOperation>
    </ComponentOperationList>

    <Configuration>
        <ComponentRelationList>
            <ComponentName>MI</ComponentName>
            <Description>MI(Call)SYS(Nest)CMS(Nest)OA(Nest)GEN</Description>
            <ComponentName>SYS</ComponentName>
            <Description>SYS(Call)CG(Nest)MG(Nest)RG</Description>
            <Description>MI(Call)SYS</Description>
            <ComponentName>CMS</ComponentName>
            <Description>CMS(Call)[NG(Contain)RG](Nest)LG(Nest)FG</Description>
            <Description>MI(Call)CMS</Description>
            <ComponentName>OA</ComponentName>
            <Description>MI(Call)OA(Nest)</Description>
            <ComponentName>GEN</ComponentName>
            <Description>MI(Call)GEN</Description>
        </ComponentRelationList>
    </Configuration>

</ComponentDescription>
```

Fig. 19 XML configuration file.

by changing the software algebraic expression of component.

Assuming a new functional module management (JG) needs to be added to the enterprise organizational structure module during the software operation, we only need to change its built-in algebraic expression of component; that is, we change CG into CG $\oplus$ JG, then the original algebraic expression of component MI $\oplus$ (SYS $\oplus$ CG $\otimes$ MG $\otimes$ RG) $\otimes$ [CMS $\oplus$ (NG $\odot$ RG) $\otimes$ LG $\otimes$ FG] $\otimes$ OA $\otimes$ GEN becomes MI $\oplus$ [SYS $\oplus$ (CG $\oplus$ JG) $\otimes$ RG] $\otimes$ [CMS $\oplus$ (NG $\odot$ RG) $\otimes$ LG $\otimes$ FG] $\otimes$ OA $\otimes$ GEN, and the corresponding configuration XML file becomes as shown in Fig. 20.

```
<ComponentDescription>
    <ComponentList>
        <ComponentName>MI</ComponentName>
        <ComponentName>SYS</ComponentName>
        <ComponentName>CMS</ComponentName>
        <ComponentName>OA</ComponentName>
        <ComponentName>GEN</ComponentName>
        <ComponentName>CG</ComponentName>
        <ComponentName>MG</ComponentName>
        <ComponentName>RG</ComponentName>
        <ComponentName>NG</ComponentName>
        <ComponentName>LG</ComponentName>
        <ComponentName>FG</ComponentName>
        <ComponentName>JG</ComponentName>
    </ComponentList>

    <ComponentOperationList>
        <ComOperation Attr="Call">Call Operation</ComOperation>
        <ComOperation Attr="Nest">Nest Operation</ComOperation>
        <ComOperation Attr="Contain">Contain Operation</ComOperation>
    </ComponentOperationList>

    <Configuration>
        <ComponentRelationList>
            <ComponentName>MI</ComponentName>
            <Description>MI(Call)SYS(Nest)CMS(Nest)OA(Nest)GEN</Description>
            <ComponentName>SYS</ComponentName>
            <Description>SYS(Call)[CG(Call)JG](Nest)MG(Nest)RG</Description>
            <Description>MI(Call)SYS</Description>
            <ComponentName>CMS</ComponentName>
            <Description>CMS(Call)[NG(Contain)RG](Nest)LG(Nest)FG</Description>
            <Description>MI(Call)CMS</Description>
            <ComponentName>OA</ComponentName>
            <Description>MI(Call)OA(Nest)</Description>
            <ComponentName>GEN</ComponentName>
            <Description>MI(Call)GEN</Description>
        </ComponentRelationList>
    </Configuration>

</ComponentDescription>
```

Fig. 20 XML configuration file after increasing the organization management.

Through the above experiment, we have proved that the evolution of software systems can be mapped to software algebraic expression of components. Changing the algebraic expression of components will result in a corresponding change in software system evolution. This method applies the algebraic theory to the process of software evolution and regards the various functional modules of the software system as different algebraic components. Three kinds of computing relationship between algebraic components, calling, nesting, and including, are used to represent the relationships between the software system functional modules to ensure a more efficient evolution of the software systems.

5 Conclusion

In this paper, we propose an algebraic topology-based method for studying dynamic software systems, whereby any arbitrary software system can be abstracted into an algebraic expression of components, which is then used to denote software system evolution.

In this paper, a software is considered as a kind of engineering system that implements logic control, and its function modules can be abstracted into the representation of different algebraic components. This paper also presents a method of abstracting any software system into a topology diagram of software structure, which can express the source codes of software system. In addition, a method for transforming the topology diagram into an algebraic expression of components is presented, and the algebraic component of the software system is represented by $\mathrm{PM} = (X, Y, D, C, \tau, \eta)$. The relationship between the algebraic components is divided into three operational relationships: calling, nesting, and including. It was proved that the calling, nesting, and including operational relationships between algebraic components can form a complete algebraic system, and these three operational relationships help to abstract any arbitrary software system into an algebraic expression of components.

The process of software system evolution was analyzed from three aspects: addition of algebraic components, deletion of algebraic components, and modification of algebraic components, and a corresponding evolutionary algorithm was proposed.

A new research method was developed to analyze the software system evolution based on algebraic topology. This method pioneers a new field of using algebraic theory to analyze software system evolution and shifting the research on software system to center on algebraic expressions; this is expected to further develop in the future.

Acknowledgment

This work was supported by the National Natural Science Foundation of China (No. U1636115) and the National Key R&D Program of China (No. 2016YFB0800700).

References

[1] H. P. Breivold, I. Crnkovic, and M. Larsson, A systematic review of software architecture evolution research, *Information & Software Technology,* vol. 54, no. 1, pp. 16–40, 2012.

[2] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos, Graph-based analysis and prediction for software evolution, presented at the 34th International Conference on Software Engineering, Zurich, Switzerland, 2012.

[3] R. D. Cosmo, D. D. Ruscio, P. Pelliccione, A. Pierantonio, and S. Zacchiroli, Supporting software evolution in component-based FOSS systems, *Science of Computer Programming,* vol. 76, no. 12, pp. 1144–1160, 2011.

[4] H. L. Chen and L. I. Ren-Fa, Dynamic evolution mechanism oriented to service-object, *Journal of Computer Applications,* vol. 30, no. 7, pp. 1974–1977, 2010.

[5] F. Dai, T. Li, Z. W. Xie, Q. Yu, and P. Lu, Towards an algebraic semantics of software evolution process models, (in Chinese), *Journal of Software*, vol. 23, no. 4, pp. 846–863, 2012.

[6] Keira, Common Object Request Broker Architecture (CORBA), https://www.ibm.com/support/knowledgecenter/SSMKHH_10.0.0/com.ibm.etools.mft.doc/bc22400_.htm, 2015.

[7] Z. Onderka, DCOM and CORBA efficiency in the wireless network, *Computer Networks*, vol. 291, pp. 448–458, 2012.

[8] W. Darwish and K. Beznosov, Analysis of ANSI RBAC support in EJB, *International Journal of Secure Software Engineering,* vol. 2, no. 2, pp. 25–52, 2011.

[9] F. Q. Yang, H. Mei, and K. Q. Li, Software reuse and software component technology, (in Chinese), *ACTA ELECTRONICA SINICA,* vol. 2, no. 27, 1999.

[10] Y. S. Zhang and X. Li, Design method of software architecture based on component operation, (in Chinese), *Computer Engineering,* vol. 34, no. 9, pp. 48–49, 2008.

[11] W. Cazzola and A. Shaqiri, Dynamic software evolution through interpreter adaptation, presented at the 15th International Conference on Modularity, ACM, Mlaga, Spain, 2016.

[12] X. Sun, Y. Chai, Y. Liu, J. Shen, and Y. Huang, Evolution of specialization with reachable transaction scope based on a simple and symmetric firm resource allocation model, *Tsinghua Science and Technology,* vol. 22, no. 1, pp. 10–28, 2017.

[13] R. Jiang and M. Yang, Survey on software complexity research, *Computer Systems & Applications,* vol. 23, no. 9, pp. 1–5, 2014.
[14] C. Grabow, S. Grosskinsky, and M. Timme, Small-world network spectra in mean-field theory, *Physical Review Letters,* vol. 108, no. 21, p. 218701, 2012.
[15] A. Shaukat and J. P. Thivierge, Statistical evaluation of waveform collapse reveals scale-free properties of neuronal avalanches, *Frontiers in Computational Neuroscience,* vol. 10, no. 163, 2016.
[16] Y. Liu, J. J. Slotine, and A. L. Barabasi, Controllability of complex networks, *Nature,* vol. 473, no. 7346, pp. 167–173, 2011.
[17] D. Chen, L. Lü, M. S. Shang, Y. C. Zhang, and T. Zhou, Identifying influential nodes in complex networks, *Physical A Statistical Mechanics & Its Applications,* vol. 391, no. 4, pp. 1777–1787, 2012.
[18] Z. Liu, T. Li, X. Yu, and X. Wang, The verification analysis of the software dynamic evolution topology structure model based on demand and runtime variability parallel driver under the background of large data, presented at the 6th International Conference on Machinery, Materials, Environment, Biotechnology and Computer, Tianjin, China, 2016.
[19] C. Chen, X. H. Hu, K. Zheng, X. Wang, Y. Xiang, and J. Li, HBD: Towards efficient reactive rule dispatching in software-defined networks, *Tsinghua Science and Technology,* vol. 21, no. 2, pp. 196–209, 2016.
[20] J. Ruths and D. Ruths, Control profiles of complex networks, *Science,* vol. 343, no. 6177, pp. 1373–1376, 2014.
[21] X. Q. Peng, X. D. Yan, and J. X. Wang, Framework to identify protein complexes based on similarity preclustering, *Tsinghua Science and Technology,* vol. 22, no. 1, pp. 42–51, 2017.

Chun Shan received the PhD degree in computer science from Beijing Institute of Technology in 2015. She is a lecturer and master tutor of School of Software in Beijing Institute of Technology. Her research interests include software security, network security, and artificial intelligence. Now she is leading the project "Software Vulnerability Detection Methods and Techniques Based on Topological Invariant" supported by the National Natural Science Foundation of China (No. U1636115).

Liyuan Liu is currently a master student of school of software in Beijing Institute of Technology. She received the bachelor degree from Beijing Institute of Technology in 2016. Her research interests are software security and artificial intelligence.

Jingfeng Xue received the PhD degree in computer science from Beijing Institute of Technology in 2003. He is the vice dean, professor, and doctoral supervisor of School of Software in Beijing Institute of Technology. His research interests include cyberspace security and artificial intelligence.

Changzhen Hu received the PhD degree in information security from Beijing Institute of Technology in 1996. He is the vice dean, professor, and doctoral supervisor of School of Software in Beijing Institute of Technology. He led the project of the National Key R&D Program of China (No. 2016YFB0800700). His research interest is cyberspace security.

Hongjin Zhu received the master degree in software engineering from Beijing Institute of Technology in 2017. He is currently a software engineer in Shenzhen. His research interest is software security.