Towards a Formal Approach to Defining and Computing the Complexity of Component Based Software

Yongxin Zhao, Xiujuan Zhang
Shanghai Key Laboratory of Trustworthy Computing,
East China Normal University, Shanghai, China
Email:yxzhao@sei.ecnu.edu.cn, 51164500266@stu.ecnu.edu.cn

Ling Shi
School of Information Systems,
Singapore Management University, Singapore
Email:lshi@smu.edu.sg

Gan Zeng, Feng Sheng
Shanghai Key Laboratory of Trustworthy Computing,
East China Normal University, Shanghai, China
Email:10152510281@stu.ecnu.edu.cn, fsheng1990@163.com

Shuang Liu
School of Computer Software,
Tianjin University, Tianjin, China
Email:shuang.liu@tju.edu.cn

***Abstract*—With the rapid development of software engineering and the widely adoption of software systems in various domains, the requirement for software systems is becoming more and more complex, which results in very complex software systems. Motivated by the principle of divide and conquer, component based software development is an effective way of managing the complexity in software development. In this paper, we propose a calculus to formally describe the functional and performance specification of component based software and provide formal semantics for the proposed calculus. Then we provide a method to measure the dynamic complexity of software compositions based on the proposed calculus. Finally, we define a set of algebraic laws to manifest the complexity relations between different functionally equivalent components. We conduct a case study with a real software system and the results show that our method is able to calculate the dynamic complexity of component based systems, and the complexity can be reduced based on our algebraic laws.**

***Index Terms*—Software complexity; Complexity semantics; Complexity computation; Algebraic laws**

I. Introduction

With the rapid development of software engineering, the scale of software systems is becoming larger and the complexity is becoming higher [1]. In order to meet the requirements of long-term reliable software, the ability to understand and measure the complexity of the software system is becoming increasingly important. It is extremely important in the fields of aviation, aerospace and railway transportation It is an important and decisive factor in the success or failure of a system [2], and the development of software systems can be highly risky when software complexity is neglected. Moreover, the higher the complexity of a software, more likely to introduce errors during development and maintenance. Therefore, being able to measure the complexity of a software quantitatively may help in software development and maintenance activities, especially for component based software development (CBSD) [3], which is an effective method to manage complex software development.

There are many methods of software complexity metrics, such as LOC code line measurement [4], Halstead method [5], McCabe Cyclomatic complexity measurement method [6], and CK measurement method [7] proposed in the literature. These methods are able to measure the complexity of the software products to some extent. However, they suffer from some limitations, i.e., the LOC code line metric ignores the structure and the different complexity of each code line of the software [8], and it only simply predicts and evaluates the performance and quality of the software from the number of lines of code. The McCabe Cyclomatic complexity ignores the complexity from the data flow of the software [8]. Furthermore, these methods describe the static complexity of the software and cannot describe the dynamic complexity. Dynamic complexity captures the complexity of software during execution, which is a more accurate way to understand the runtime complexity, and thus the resource consumptions of a system. Moreover, the research on the software complexity of component focuses on the analysis at the structural level only [9], and it does not consider the calculation at the semantic level. Therefore, we are motivated to propose a method to measure the dynamic complexity of a software system.

Zuse [10] provides a definition of software complexity which is the difficulty degree of software maintenance, change and understanding. This is also known as cognitive complexity. And Shao and Wang [11] also propose cognitive complexity. It provides a more comprehensive way of describing complexity and removing the effect of unrelated factors. In cognitive informatics, the functional complexity of software design and development depends mainly on three elements: input, internal processing, output. Misra [12] also proposes cognitive complexity which takes into account both the internal architecture of software and I/Os it processes. However, these methods

2640-0715/19/$31.00 ©2019 IEEE
DOI 10.1109/APSEC48747.2019.00052

can only describe the static complexity and cannot describe the dynamic complexity. Therefore, based on the idea of cognitive complexity, we propose a new method, which can measure the dynamical complexity of software.

Component based software development (CBSD) [3] is a common method of software development. Under the CBSD, the pattern of software development has changed correspondingly, i.e., the software development is not based on the algorithm and the data structure, but based on the architecture of component development and the composition. It shifts the focus of software development from program compilation to composition based on existing components. Using CBSD method can improve the efficiency of software development, improve the reusability of components, and reduce the cost of development [13].

In this work, we are motivated to propose a new computational cognitive complexity method based on CBSD method. During the execution of the component composition, we not only consider the complexity of resource consumption, but also consider the complexity of the combination so that we can calculate the dynamic complexity. We define our dynamic complexity based on execution traces of components. Before computing the complexity, we define a language to formally model the component composition and software structure in the component-based software development. Our specification language can not only describe the function of the component execution, but also describe its performance. In addition, we provide the operational semantics for the language, and propose a set of algebraic laws which enables reducing complexity in component composition. The main contributions of our work include:

- **Complexity Component Calculus**. We propose a new language called complexity component calculus for component based software development. Our language describes both the structure and the complexity of a software system. Meanwhile, we also provide a graphical representation of our language to describe the software structure and help designers understand and describe the software structure more clearly.
- **The Complexity Semantics**. We formally define the operational semantics of component composition for the proposed language. The operational semantics specifies the rules of components execution and complexity produced. According to these rules, we can infer which components have already consumed large amounts of resources.
- **The Complexity Computation**. We define a method to calculate the dynamic complexity of component executions. In addition, we also propose a set of algebraic laws for component compositions, which enables reducing complexity by functionally equivalent or functionally refinement transformations.

The rest of the paper is organized as follows. Section II defines complexity component calculus. Section III describes the execution semantics of component combinations. Section IV defines the calculation of the complexity of software structure. Section V lists some algebraic rules to discuss the optimization of system design. Section VI shows how to apply our method with a real world case study. We conclude the paper and provide future directions in Section VII.

II. The Complexity Component Calculus

In this section, we present a complexity component calculus to describe the (functional and performance) specification of component based software. In our work, we mainly consider the software complexity of performance specification.

In traditional modeling methods, an activity is generally regarded as a basic task in order to fulfill a functional operation of software. Here, we introduce *complexity activity* to extend the activity by adding the complexity information, which describes not only the functionality of an activity but also the complexity of the activity. A component is composed of a sequence of complexity activities. The formal definition of a complexity activity is given as follows:

Definition 1 (Complexity Activity)**.** *A complexity activity $\vec{a}$ is defined as a tuple (f, c), which consists of*

- *a functionality activity f, which describes the functional behavior of the complexity activity $\vec{a}$. In general, we use symbol $\mathbb{F}$ to define the set of all possible functionalities.*
- *a complexity of activity c ($c \geq 1$) , which indicates the degree of difficulty or relative time and effort required for performing the given complexity activity. We specify the minimum complexity c is 1.*

The complexity activity is the smallest unit of a component, and a complex activity can be a component by itself, that is atomic complexity component. To calculate the complexity of a structure, we first define complexity of every activity, and based on the sequence of activities of a component, we can compute complexity of the component. Then according to the complexity of each component and the combination relation between components, we can compute complexity of a structure.

We define projection functions $\pi_f(\vec{a})$ and $\pi_c(\vec{a})$ to obtain the functionality and complexity of complexity activity $\vec{a} = (f, c)$ respectively, i.e., $\pi_f(\vec{a}) = f$ and $\pi_c(\vec{a}) = c$.

Besides, we introduce a *(complexity) multiply* to define a new complexity activity, which changes the complexity of the activity performing the same functionality and the definition is listed below:

Definition 2 (Complexity Multiplication)**.** *Given a complexity activity $\vec{a} = (f, c)$ and a real number $t > 0$, a complexity multiplication $t \cdot \vec{a}$ (or $t\vec{a}$) declares a new complexity activity such that $t\vec{a} = (f, tc)$, $tc \geq 1$.*

For complexity multiplication $t\vec{a} = (f, tc)$, if the $tc < 1$, the activity $t\vec{a}$ is undefined, because we specify a minimum complexity is 1.

Two different complexity activities $\vec{a}_1 = (f_1, c_1)$ and $\vec{a}_2 = (f_2, c_2)$ may have the same functionality but different complexity, which is characterized by the formula $\pi_f(\vec{a}_1) = \pi_f(\vec{a}_2) \wedge \pi_c(\vec{a}_1) \neq \pi_c(\vec{a}_2)$.

In particular, we define a particular activity $\vec{\infty} = (f, \infty)$ to indicate that the component has entered an abnormal state. The function of this activity is invalid, and the component does not perform it, but its complexity is infinity. This activity occurs when the component encounters an exception.

In the sequel, we propose a complexity component calculus to describe the specification of a component based software using our defined complexity activities. Where P and Q represent complexity components, and $\vec{a}$ is the basic unit of the complexity component, the $t \cdot P$ is the multiplication, $P \uparrow_A$ represents restriction, $P[\vec{b}/\vec{a}]$ indicates rename, $P;Q$, $P\|_l Q$, $P\Box_l Q$ and $P^{\curvearrowright l}Q$ represent sequential, parallel, choice and call respectively. This syntax mainly describes the composition rules and the relationship between components. The abstract syntax of complexity components is defined in the following:

$$P, Q \quad ::= \quad \vec{a} \mid t \cdot P \mid P \uparrow_A \mid P[\vec{b}/\vec{a}] \mid P;Q \mid P\|_l Q \mid P\Box_l Q \mid P^{\curvearrowright l}Q$$

Where $t > 0$, $l \geq 1$, $t, l \in R^+$, and $A \subseteq \mathbb{F}$.

For each complexity component P, we employ Σ_P named *(functionality) alphabet* to represent the set of functionalities which can be performed by P. Obviously we have $\Sigma_P \subseteq \mathbb{F}$. Σ_P can be further categorized into two types, i.e., L_P and C_P, and $L_P \cup C_P = \Sigma_P$. L_P is the set of local functionalities, which are functionalities that are performed independently. C_P is the set of communicating functionalities, which are functionalities that need to be performed in coordination with other components. In our work, we employ *complexity component diagrams* to depict how atomic complexity components are composed together to form larger complexity components and ultimately software systems. The diagram of a complexity component P is shown in Fig. 1 *component*. The square represents a component, the center symbol P of the square is name of component, and the left-bottom symbol c of the square is the complexity of P. The top circle of square indicates the interface of communication. C_P represents the set of communications. In general, we can omit the alphabet in the diagram if it is not mentioned deliberately.

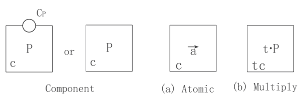

Fig. 1. The diagrams of a Component, Atomic and Multiplication.

Then we will give an informal interpretation of atomic complexity components and composed complexity components. Complexity component $\vec{a}$ in Fig. 1 (a) indicates the component performs its functionality with complexity $\pi_c(\vec{a}) = c$. Multiplication $t \cdot P$ in Fig. 1 (b) has the same functional behaviors with component P except that the complexity is t times more than that of P.

Restriction $P \uparrow_A$ in Fig.2 (a) indicates that the component P can only execute functionalities in set A. Note that any communicating functionality out of set A will not be executed. The collection A is the communicating functionalities of component P. Rename $P[\vec{b}/\vec{a}]$ in Fig. 2 (b) replaces all occurrence of $\vec{a}$ in component P with activity $\vec{b}$. The diagrams of the two components are shown in Fig. 2.

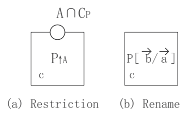

Fig. 2. The diagrams of restriction and rename.

The sequential operator $P;Q$ in Fig. 3 (a) indicates the sequential execution of components P and Q. After component P is executed, component Q will begin to execute. In addition, the sequential operator will not bring in more complexity. The parallel operator $P\|_l Q$ in Fig. 3 (b) indicates that component P and Q execute in parallel and they can cooperate on common functionalities with each other. Obviously, parallel will lead to higher complexity due to the cooperation of P and Q. Therefore, we use parameter l to represents the complexity of parallel relation operator. So l should be greater than 1.

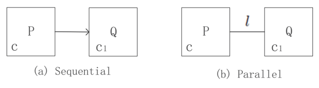

Fig. 3. The diagrams of sequential and parallel.

Similarly, the choice operator and the call operator also bring in more complexity than simply adding up the workload of their composing components. The choice operator $P\Box_l Q$ in Fig. 4 (a) is resolved non-deterministically, i.e., two components P and Q are selected to be executed by either the environment or the system. In the Fig. 4 (a), we use a dashed line to connect two components to represents choice operator, and in Fig. 3 (b), we use a solid line to connect two components to represents parallel operator. The call $P^{\curvearrowright l}Q$ in Fig. 4 (b) indicates that the execution of the P component will be suspended if the invoking of component Q occurs, and after the Q component is finished, component P resumes its execution. We use dashed arrow to connect two components to represents call operator, while use solid arrow to connect two components to represents sequential operator in Fig. 3 (a).

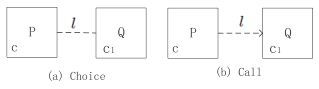

Fig. 4. The diagrams of choice and call.

In addition, we use the dotted box, which is similar to the function of parentheses adopted in mathematics, mainly to define the execution sequence of the component compositions. The compositions in the dashed box have higher priorities. Fig. 5 shows an example diagram of component $P^{\curvearrowright 3}(Q\|_4 R)$.

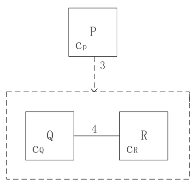

Fig. 5. An example of component execution priority.

III. The Complexity Semantics

In this section, we define the complexity semantics for complexity component calculus. The semantics specifies not only the functional behavior of the transition performed, but also the complexity calculation incurred during the composition.

Before defining the complexity semantics, we employ two special states ϕ and $\perp$ to describe the possible execution states of the complexity components. State ϕ indicates the normal termination of component while state $\perp$ represents the deadlock state. Our complexity semantics is defined by transitions between *configurations*. A configuration is either a complexity component or a special state. We use symbol $\mathbb{C}^{\{\perp,\phi\}}$ to represent the set of all configurations.

Our complexity semantics is defined with the help of a structural equivalence, which defines equivalent components. The structural equivalence formulas of components are given in Table I.

Table I. The Structural Equivalence

$\phi; Q = Q$	$\perp; Q = \perp$	$P \parallel_l Q = Q \parallel_l P$
$P \parallel_l \phi = P$	$\perp \parallel_l \perp = \perp$	$\phi \curvearrowright^{l} Q = \phi$
$\perp \curvearrowright^{l} Q = \perp$	$P \curvearrowright^{l} \perp = \perp$	$P \curvearrowright^{l} \phi = P$

Definition 3 (Transition Rule)**.** *Given the complexity components P, P' and an activity $\vec{a}$, the transition rule is defined as $P \xrightarrow{\vec{a}} P'$.*

The transition rule describes that when component P executes an activity $\vec{a}$, its state transfers from P to P'.

Table II shows the formal operational semantics (with complexity) of the complexity component calculus operators.

The Multiplication rule denotes that given the rule $P \xrightarrow{\vec{a}} P_1$, component $t \cdot P$ executes activity $t\vec{a}$ with complexity $t\pi_c(\vec{a})$ and the subsequent component is $t \cdot P_1$. The multiplication rule indicates that component $t \cdot P$ just only changes the complexity of the activity and executes the same functionality with component P. Thus the introduction of multiplication $t \cdot P$ provides a mechanism of describing components with the same functionalities and different complexity.

Sequential combination is one of the most common combinations of software design, which captures the sequential execution order of the components. In general, the sequential operation $P; Q$ executes component P first, and then executes component Q after P is completed. The seq-1 rule indicates that the component P is executed, and component Q cannot be executed before the execution of component P is finished.

Table II. The operational semantics of complexity calculus

(Multiplication)	$\dfrac{P \xrightarrow{\vec{a}} P_1}{t \cdot P \xrightarrow{t\cdot\vec{a}} t \cdot P_1}$	(Seq-1)	$\dfrac{P \xrightarrow{\vec{a}} P_1}{P; Q \xrightarrow{\vec{a}} P_1; Q}$
(Seq-2)	$\dfrac{P \xrightarrow{\vec{a}} \phi}{P; Q \xrightarrow{\vec{a}} Q}$	(Choice-L)	$\dfrac{P \xrightarrow{\vec{a}} P_1}{P \Box_l Q \xrightarrow{l\cdot\vec{a}} P_1}$
(Choice-R)	$\dfrac{Q \xrightarrow{\vec{a}} Q_1}{P \Box_l Q \xrightarrow{l\cdot\vec{a}} Q_1}$	(Call-1)	$\dfrac{P \xrightarrow{\vec{a}} P_1}{P \curvearrowright^{l} Q \xrightarrow{\vec{a}} P_1 \curvearrowright^{l} Q}$
(Call-2)	$\dfrac{Q \xrightarrow{\vec{a}} Q_1}{P \curvearrowright^{l} Q \xrightarrow{l\cdot\vec{a}} (l \cdot Q_1); P}$	(Rename)	$\dfrac{P \xrightarrow{\vec{a}} P_1}{P[\vec{b}/\vec{a}] \xrightarrow{\vec{b}} P_1[\vec{b}/\vec{a}]}$
(Restriction)	$\dfrac{P \xrightarrow{\vec{a}} P_1, \pi_f(\vec{a}) \in A \cap C_P}{P{\uparrow_A} \xrightarrow{\vec{a}} P_1{\uparrow_A}}$		
(Parallel-L)	$\dfrac{P \xrightarrow{\vec{a}} P_1,\ \pi_f(\vec{a}) \notin C_Q}{P \parallel_l Q \xrightarrow{l\cdot\vec{a}} P_1 \parallel_l Q}$		
(Parallel-R)	$\dfrac{Q \xrightarrow{\vec{a}} Q_1, \pi_f(\vec{a}) \notin C_P}{P \parallel_l Q \xrightarrow{l\cdot\vec{a}} P \parallel_l Q_1}$		
(Parallel-Co)	$\dfrac{P \xrightarrow{\vec{a}_1} P_1, Q \xrightarrow{\vec{a}_2} Q_1, \pi_f(\vec{a}_1) = \pi_f(\vec{a}_2) \in C_P \cap C_Q}{P \parallel_l Q \xrightarrow{l^2\cdot\pi_c(\vec{a}_1)\cdot\vec{a}_2} P_1 \parallel_l Q_1}$		

Rule seq-2 represents when P component finishes, component Q starts to execute.

The choice combination $P \Box_l Q$ is employed to deal with the situation that the system may execute component P or component Q. The two components cannot be executed concurrently. Rule choice-L indicates that the system chooses to execute component P while in rule choice-R, component Q is selected to be executed. The activity $l \cdot \vec{a}$ is performed to indicate that the added complexity caused by the choice is considered in the execution of the choice combination.

Next, the behavior of the call combination is investigated, which is demonstrated by transition rules Call-1 and Call-2 in Table II. The call $P \curvearrowright^{l} Q$ is a particular combination of software components, in which the call may occur and make an influence on the execution of P. The Call-1 rule indicates that the component P would execute to the end if the call does not occur; the Call-2 rule shows that at some point, the execution of component P would be suspended and component Q starts to execute. P will continue the execution once the call operation ends.

For restriction, $P{\uparrow_A}$ indicates that P can execute an activity normally if the activity belongs to $A \cap C_P$. The formula restriction in Table II indicates that when P encounters an activity in $A \cap C_P$, it can execute the activity and transit to another state normally. In other words, P automatically ignores the activities that are not in $A \cap C_P$, remains in the original state, and waits for the next activity to be executed.

Rule Rename represents that all the concurrences of $\vec{a}$ will be replaced by $\vec{b}$, and other activities are executed normally. The complexity of this step is the complexity of $\vec{b}$.

$P \parallel_l Q$, which indicates that the P and Q components are performed in parallel. The execution order of the two components is nondeterministic, that is, the execution order of the components is not cared for during the execution. In addition, when performing parallel operations, synchronization is determined by the function of the activity in the two components. If the two components don't have the same active function, synchronization will not occur. However, if they have the same function, synchronization will occur. If the synchronization activity does not occur, the component performs its own activities without communicating with other components, we use $l \cdot \pi_c(\vec{a})$ to represent the complexity of such an execution. When synchronization activity occurs, components are required to execute simultaneously, and we use $l^2 \cdot \pi_c(\vec{a}_1) \cdot \pi_c(\vec{a}_2)$ to indicate the complexity of synchronous execution. This is demonstrated by the transition rules Parallel-L, Parallel-R and Parallel-Co in Table II.

As can be seen from the table above, in the process of the components' execution, Parallel-L and Parallel-R capture the situation the common events of P and Q do not occur, and thus no real synchronization happens. When an exception state occurs during the execution component P, it enters this state: $\dfrac{P \xrightarrow{\vec{a}} \perp,\ \pi_f(\vec{a}) \notin C_Q}{P \parallel_l Q \xrightarrow{l \cdot \vec{a}} \perp \parallel_l Q}$, that is, component P enters an abnormal state at the execution time.

The Parallel-Co formula captures the situations that the activity occurs in both component P and Q, and thus a synchronous execution of P and Q is required. During the synchronized execution, if one of the components encounters an exception, the synchronization cannot proceed, the execution semantics rule is:

$$\frac{P \xrightarrow{\vec{a}_1} \perp,\ Q \xrightarrow{\vec{a}_2} Q_1,\ \pi_f(\vec{a}_1) = \pi_f(\vec{a}_2) \in C_P \cap C_Q}{P \parallel_l Q \xrightarrow{l^2 \cdot \pi_c(\vec{a}_1) \cdot \vec{a}_2} \perp}$$

This formula shows that during the synchronization operation, any component abnormality will make the combination enter an exception state.

We also consider the deadlock situations in synchronization. Example 1 illustrates the deadlock situation.

Example 1. Given that parallel component $P \parallel_l Q$, where $P = \vec{a}; \vec{b}; \vec{c}$, and $Q = \vec{d}; \vec{c}; \vec{b}$. After activities $\vec{a}$ and $\vec{d}$ are performed, component P is transited to P_1 and Q is transited to Q_1, that is, $\dfrac{P \xrightarrow{\vec{a}} P_1}{P \parallel_l Q \xrightarrow{l \cdot \vec{a}} P_1 \parallel_l Q}$, $\dfrac{Q \xrightarrow{\vec{d}} Q_1}{P_1 \parallel_l Q \xrightarrow{l \cdot \vec{d}} P_1 \parallel_l Q_1}$.

When executing the next step, P_1 would execute $\vec{b}$ which needs to synchronize with Q_1. But Q_1 needs to perform $\vec{c}$ first, which also needs to synchronize with P_1. Therefore, P_1 and Q_1 are unable to execute because they are waiting for each other. This is a typical deadlock situation which may occur for concurrent components.

In order to describe the deadlock situation, we introduce predicates $Disable(P, A)$, which represents that P is unable to execute any functionality a in set A. The formal definition is given below.

Definition 4 (Disable)**.** *Given that complexity component P and functionality set A ($A \subset \Sigma_P$), we say component P is disable in A if for any functionality a in set A, P could not execute a with a finite complexity. We denote it by predicate $Disable(P, A)$, i.e.,*

$$\begin{aligned} Disable(P, A) \quad =_{df} \quad & \forall a \in A, \exists \vec{a} \bullet \pi_f(\vec{a}) = a \\ & \wedge \neg(\exists P_1, P \xrightarrow{\vec{a}} P_1) \end{aligned}$$

And in $Disable(P, A)$, the set A represents that collection of activities in Σ_P. Particularly, $Disable(P)$ indicates that P cannot perform any activity. When a component encounters $disable(P, a)$ at execution time, it suspends execution, and remains in the previous state.

In the above, we adopt a single-step transition to describe the behavior of components. Now, we lift the transition to allow the system executing a sequence of activities.

Definition 5 (Sequence Transition)**.** *Given that components P, P' and a sequence of activities $\vec{s}$, the sequence transition $P \xrightarrow{\vec{s}} P'$ is recursively defined as follows:*

$$P \xrightarrow{\vec{s}} P' =_{df} \begin{cases} P \xrightarrow{\vec{a}} P', & |\vec{s}| = 1 \wedge \vec{s} = \vec{a} \\ (\exists P'') \bullet (P \xrightarrow{\vec{s'}} P'') \wedge & \\ (P'' \xrightarrow{\vec{a}} P'), & \vec{s} = \vec{s'} ^\frown \vec{a} \end{cases}$$

According to the transition rules defined above, a component may transit into a divergent state, which can be an infinite loop. In this case, we define a predicate $P \uparrow$ to indicate that this component is executing indefinitely.

Definition 6 (Divergent)**.** *Given that component P, we say component P is divergent if for any integer n, P could execute sequence $\vec{s}$ with length n. Formally, i.e., $\forall n, \exists \vec{s}, |\vec{s}| = n, \exists P' \bullet P \xrightarrow{\vec{s}} P'$.*

Therefore, when we intend to compute the complexity of component which is divergent, we can only consider the first n steps that the component performs if the complexity of the component is measured within n transition steps.

IV. The Complexity Computation

In this section, we formally define formulas to calculate the complexity of component compositions. Our approach calculates dynamic complexity, which is based on the execution traces collected dynamically in the component composition process. We also show how to estimate the overall dynamic complexity with average complexity when nondeterminism occurs. Compared to the manual complexity computation based on software testing, our approach could traverse all possible execution paths.

We first define execution trace $Tr(P)$, which is based on a sequence of activities performed during the component composition, as follows:

Definition 7 (Execution Trace)**.** *Given a component P, we define $Tr(P)$ to represent the sequence of activities performed, which is formally defined below:*

$$\begin{aligned} Tr(P) \quad = \quad & \{\vec{s} \mid P \xrightarrow{\vec{s}} \phi\} \cup \{\vec{s} \cdot \vec{\infty} \mid P \xrightarrow{\vec{s}} \perp\} \cup \\ & \{\vec{s} \mid \exists P' \bullet P \xrightarrow{\vec{s}} P' \wedge P' \uparrow\} \cup \\ & \{\vec{s} \mid \exists P' \bullet P \xrightarrow{\vec{s}} P' \wedge Disable(P')\} \end{aligned}$$

This formula represents a collection of all possible traces that a component executes. The first situation indicates that the program terminates normally, and the component performs $|\vec{s}|$ steps. The second situation indicates the component encounters an exception, and the complexity is infinity. The third situation illustrates infinite steps in the execution of the component and the path is divergent, so we chose all possible activity paths $\vec{s}$ which lead to divergence as its execution trace. The last situation indicates that the component is stuck in a state that any activity cannot be performed after the component executes $|\vec{s}|$ steps, and the component is disable.

The above formula records the possible traces of complexity activities performed throughout the execution of the component, including the path in an exception state and the path that the component performs infinite steps.

Based on the information of the trace that the component executes, we are able to get a collection of activities performed by the component. Now we define how to get the functionality TF of trace. The formula is:

$$TF(\vec{s}) =< f_1, f_2, ..., f_n >, f_i = \pi_f(\vec{s}(i)), 1 \leq i \leq n, \ |\vec{s}| = n$$

where $\vec{s}(i)$ indicates the ith activity in a sequential activity $\vec{s}$.

With all the previous definitions, we define how to calculate trace complexity TC for a given trace $\vec{s}$. The formula is:

$$TC(\vec{s}) = \sum_{i=1}^{n} \pi_c(\vec{s}(i)), \ |\vec{s}| = n$$

Where $\vec{s}(i)$ indicates the ith activity in $\vec{s}$. According to this formula, we can represent the complexity of any finite sequential activities in the execution process. However, when the length of $\vec{s}$ is infinite, we also can select the sequence of its first n activities to computing the complexity. The formula is as follows:

$$TC(\vec{s}, n) = \sum_{i=1}^{n} \pi_c(\vec{s}(i)), \ |\vec{s}| > n$$

However, there is a special complexity, for those software structures that reach an exception states during execution, we define its complexity as infinity, which indicates an error in the component and modification is required.

When the sequence activities of the component execution is determined, its complexity is a definite value. However, the execution path of some components is nondeterministic, such as choice combination, the path is chosen based on the environment. At this point, we introduce the average complexity to estimate the complexity of such component combination, we use $AC(P)$ to estimate the general complexity.

For finite length traces, the average trace complexity is defined in the following:

$$AC(P) = \frac{1}{N} \sum_{\vec{s} \in Tr(P)} TC(\vec{s}), \ |Tr(P)| = N$$

When the length of $\vec{s}$ is infinite, the average complexity is defined as:

$$AC(P, n) = \frac{1}{N} \sum_{\vec{s} \in Tr(P)} TC(\vec{s}, n), \ |Tr(P)| = N$$

The complexity of components can be computed according to the formula above, where N is the number of all execution traces during the execution of the component.

Because there are many execution traces of some components, we use its average complexity as the complexity of the component. When measuring the complexity of a software structure, the average complexity is also used as the general complexity.

V. Algebraic Laws

In this section, we define a set of algebraic laws which holds for the complexity computation. These algebraic laws explore the relations (equations or inequations) between structure functionality and performance.

In the following, we define 7 set of laws, which caters both functionality and complexity. In the following definitions, P_1, P_2, P_3 represent three components respectively. The labels of laws starting with "L-F-" represent the laws for functionality, the labels of laws starting with "L-C-" represent the laws for complexity.

L-F-1: $((P_1; P_3)\Box_l(P_2; P_3)) =_f ((P_1 \Box_l P_2); P_3)$

L-C-1: $((P_1; P_3)\Box_l(P_2; P_3)) \geq_c ((P_1 \Box_l P_2); P_3)$

The above two laws compare the functionality and performance of the software structures composed of the sequential and the choice operator respectively. We can see from law L-F-1 that the structure on both sides of the functionality is equal, and law L-C-1 indicates that the complexity of the right structure is less than the complexity of the left structure.

L-F-2: $((P_1 \curvearrowright^{l_1} P_3)||_{l_2}(P_2 \curvearrowright^{l_1} P_3)) \sqsubseteq_f ((P_1||_{l_2}P_2) \curvearrowright^{l_1} P_3)$

L-C-2: $((P_1 \curvearrowright^{l_1} P_3)||_{l_2}(P_2 \curvearrowright^{l_1} P_3)) \geq_c ((P_1||_{l_2}P_2) \curvearrowright^{l_1} P_3)$

L-F-3: $((P_1 \curvearrowright^{l} P_3); (P_2 \curvearrowright^{l} P_3)) \sqsubseteq_f ((P_1; P_2) \curvearrowright^{l} P_3)$

L-C-3: $((P_1 \curvearrowright^{l} P_3); (P_2 \curvearrowright^{l} P_3)) \geq_c ((P_1; P_2) \curvearrowright^{l} P_3)$

L-F-4: $((P_1 \curvearrowright^{l_1} P_3)\Box_{l_2}(P_2 \curvearrowright^{l_1} P_3)) =_f ((P_1 \Box_{l_2} P_2) \curvearrowright^{l_1} P_3)$

L-C-4: $((P_1 \curvearrowright^{l_1} P_3)\Box_{l_2}(P_2 \curvearrowright^{l_1} P_3)) \geq_c ((P_1 \Box_{l_2} P_2) \curvearrowright^{l_1} P_3)$

Laws L-F-2, L-C-2, L-F-3, L-C-3, L-F-4, L-C-4 capture the functional and performance relationships between the two structures which are composed of the call operation in combination with parallel, sequential, and choice operations, respectively. We can conclude from these laws that in terms of functionality, for laws L-F-2 and L-F-3, the composition structures on the right is refine from that on the left, while for law L-F-4, the right structure is equal to that on the left. In terms of performance complexity, the composition structures on the right have better performance than those on the left.

L-F-5: $((P_1||_{l_1}P_3)\Box_{l_2}(P_2||_{l_1}P_3)) =_f ((P_1 \Box_{l_2} P_2)||_{l_1}P_3)$

L-C-5: $((P_1||_{l_1}P_3)\Box_{l_2}(P_2||_{l_1}P_3)) \geq_c ((P_1 \Box_{l_2} P_2)||_{l_1}P_3)$

Laws L-F-5, L-C-5 capture the relationship between functionality and performance in the structure composed of the parallel operator in combination with choice operator. According to the laws, we can draw a conclusion that the structure of parallel in combination with choice satisfies the law of distribution, which makes the structure on the right have an optimal complexity, yet maintains the same functionality.

L-F-6: $((P_1||_l P_3);(P_2||_l P_3)) \sqsubseteq_f ((P_1;P_2)||_l P_3)$

L-C-6: $((P_1||_l P_3);(P_2||_l P_3)) \geq_c ((P_1;P_2)||_l P_3)$

L-F-7: $((P_1 \Box_{l_1} P_3)||_{l_2}(P_2 \Box_{l_1} P_3)) \neq_f ((P_1||_{l_2} P_2)\Box_{l_1} P_3)$

L-C-7: $((P_1 \Box_{l_1} P_3)||_{l_2}(P_2 \Box_{l_1} P_3)) \geq_c ((P_1||_{l_2} P_2)\Box_{l_1} P_3)$

Laws L-F-6, L-C-6, L-F-7, L-C-7 capture the relationship between functionality and performance in the structure composed of the parallel operator in combination with sequential and choice operator. In terms of functionality, law L-F-6 indicate that the composition structures on the right is refine from that on the left, while L-F-7 represents the right structure is equal to that on the left. And in terms of performance complexity, the composition structures on the right have better performance than those on the left.

VI. Case Study

With the rapid development of computer technology, the live broadcast system is becoming more and more popular, and it is affecting different application domains, such as E-commerce, on-line education, etc. Therefore, the high performance is extremely important for live broadcast systems. In this section, we use one module of a live broadcast system, i.e., the decoding module, as a case study to illustrate our approach.

First, we refer to a domestic company's live broadcast platform system architecture diagram, based on the architecture diagram of the decoding module, we rewrite the decoding module with the component complexity calculus we propose, and use the graphical notation to represent the result decoding module, as shown in Fig. 6.

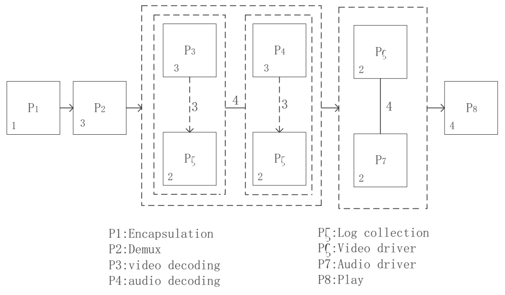

Fig. 6. The diagram of video decoding module VDM.

Fig. 6 simulates the main process of video decoding module. First, the video stream is encapsulated into video and audio streams of specified format, as is indicated by component P_1. Then the audio stream and the video stream are decoded separately by means of demux method in component P_2. After that the video stream and the audio stream are decoded concurrently, and the data packets of the corresponding format are obtained. The log collection component, which is shown captured by P_5, is called by the video/audio stream decoding component (P_3, P_4) during the decoding process. The video stream decoding file is YUV data package, and the audio stream decoding file is PCM data package. Finally, these two files are synchronized by the video driver component P_6 and the audio driver component P_7, respectively, and input to the player component P_8.

In our case study, we choose the simplest component, i.e., the encapsulation component, and set its complexity as 1. The complexities of the other components are defined as comparatively with the base component. In Fig. 6, the number on the left-bottom corner of each component square shows the complexity of the corresponding component. The numbers on the connection lines of components represent the complexity introduced by component compositions. For instance, the complexity of the parallel composition of video and audio decoding is 4, and the complexity of the call composition (from P_3 to P_5) is 3.

In Fig. 6, when the video stream and the audio stream are decoded concurrently, decoding of video stream and the audio stream can proceed separately, and no synchronization is required for decoding operations. However, when executing the decoded files to obtain the combined audio and video signals, it is required that the video images and the audio speeches are consistent. Therefore, synchronization is required for the decoded video stream file (YUV) and the decoded audio stream file (PCM).

We can represent the component composition in Fig. 6 with our component composition rules, and the result formula is: $P_1; P_2; ((P_3 \frown^3 P_5)||_4(P_4 \frown^3 P_5)); (P_6||_4 P_7); P_8$.

There are totally 16 possible execution paths for the component construction diagram shown in Fig. 6. We list all the paths and their corresponding complexities in Table III. In Table III, $path_i$ represents the ith execution path, $\vec{s_i}$ represents the sequence generated by $path_i$ and $TC(\vec{s_i})$ represents the complexity of the sequence $\vec{s_i}$.

Table III. The complexity of execution path

i	$path_i$	$TC(\vec{s_i})$
1	$P_1, P_2, P_3, P_4, P_6, P_7, P_8$	96
2	$P_1, P_2, P_3, P_4, P_7, P_6, P_8$	96
3	$P_1, P_2, P_4, P_3, P_6, P_7, P_8$	96
4	$P_1, P_2, P_4, P_3, P_7, P_6, P_8$	96
5	$P_1, P_2, P_5, P_3, P_4, P_6, P_7, P_8$	120
6	$P_1, P_2, P_5, P_3, P_4, P_7, P_6, P_8$	120
7	$P_1, P_2, P_4, P_5, P_3, P_6, P_7, P_8$	120
8	$P_1, P_2, P_4, P_5, P_3, P_7, P_6, P_8$	120
9	$P_1, P_2, P_3, P_5, P_4, P_6, P_7, P_8$	120
10	$P_1, P_2, P_5, P_4, P_3, P_6, P_7, P_8$	120
11	$P_1, P_2, P_3, P_5, P_4, P_7, P_6, P_8$	120
12	$P_1, P_2, P_5, P_4, P_3, P_7, P_6, P_8$	120
13	$P_1, P_2, P_3, P_5, P_4, P_5, P_6, P_7, P_8$	672
14	$P_1, P_2, P_4, P_5, P_3, P_5, P_6, P_7, P_8$	672
15	$P_1, P_2, P_3, P_5, P_4, P_5, P_7, P_6, P_8$	672
16	$P_1, P_2, P_4, P_5, P_3, P_5, P_7, P_6, P_8$	672

We use the first path as an example to show how to compute the path complexity with our method. The first path is:

$P_1, P_2, P_3, P_4, P_6, P_7, P_8$ and the sequence is $\vec{s_1}$. According to the data of diagram, we can compute:

$$TC(\vec{s_1}) = 1 + 3 + 4 \cdot (3 + 3) + 4^2 \cdot 2 \cdot 2 + 4 = 96$$

We can compute the complexities of the other 11 paths with the same method. The average complexity of components video decoding module (VDM) is:

$$AC(VDM) = \frac{1}{16} \sum_{i=1}^{16} TC(\vec{s_i}) = 252$$

We have shown how to represent and compute the complexity of component compositions. In the following, we show how to reduce complexity of component compositions with our proposed laws. According to Law L-F-2, the component structure in Fig. 6 can be transformed to its functional refinement format, as is shown in Fig. 7.

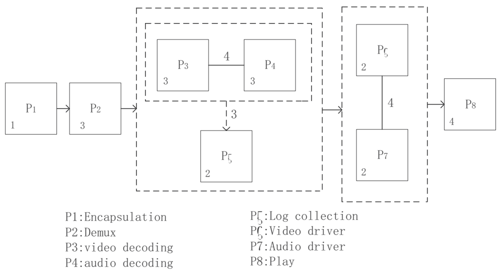

Fig. 7. The diagram of video decoding module $VDM^{'}$.

The component structure in Fig. 7 has eight possible execution paths, which are shown in Table IV.

Table IV. The complexity of execution path

i	$path_i$	$TC(\vec{s_i})$
1	$P_1, P_2, P_3, P_4, P_6, P_7, P_8$	96
2	$P_1, P_2, P_4, P_3, P_6, P_7, P_8$	96
3	$P_1, P_2, P_3, P_4, P_7, P_6, P_8$	96
4	$P_1, P_2, P_4, P_3, P_7, P_6, P_8$	96
5	$P_1, P_2, P_5, P_3, P_4, P_6, P_7, P_8$	102
6	$P_1, P_2, P_5, P_4, P_3, P_6, P_7, P_8$	102
7	$P_1, P_2, P_5, P_3, P_4, P_7, P_6, P_8$	102
8	$P_1, P_2, P_5, P_4, P_3, P_7, P_6, P_8$	102

Table IV lists all possible execution paths and the complexity of these paths. Therefore, the average complexity of new video decoding module $VDM^{'}$ is:

$$AC(VDM^{'}) = \frac{1}{8} \sum_{i=1}^{8} TC(\vec{s_i}) = 99$$

According to the computing results, we can conclude that $AC(VDM) > AC(VDM^{'})$.

Through the functional preserving transformation based on our proposed laws, we are able to make the software structure clearer, and reduce the dynamic complexity of the software. Therefore, our method is able to formally capture and systematically reduce the complexity of a software.

VII. Conclusion and Future Work

In this paper, we propose a formal language and execution semantics to describe the complexity of software structure. And on this basis, we propose the calculation method of complexity for the components executed. This method can not only compute the complexity of software structure, but also compute the dynamic complexity in the process of software components' execution. We also establish some laws to help to design a more efficient software structure.

In the future, we will explore the completeness of the components syntax, as our goal is to enable our language to represent software structures as much as possible. And we also would like to study the software structure evolution method. The evolution of software structure refers to the further optimization based on the designed structure, making the new structure more efficient.

Acknowledgement

This paper is partially supported by National Defense Basic Scientific Research Program of China (No. JCKY2016204B503), Science and Technology Commission of Shanghai Municipality Project (No. 18ZR1411600), National Natural Science Foundation of China (NSFC 61402176 and 61602177) and the Open Project of Shanghai Key Laboratory of Trustworthy Computing (No. 08dz22304201804)

References

[1] A. Sharma, R. Kumar, and P. S. Grover, "Empirical evaluation and critical review of complexity metrics for software components," in Proceedings of the 6th WSEAS International Conference on Software Engineering, Parallel and Distributed Systems, 2007.

[2] D. S. Kushwaha and A. K. Misra, "Improved cognitive information complexity measure: A metric that establishes program comprehension effort", SIGSOFT Softw. Eng. Notes, vol. 31, no. 5, pp. 1–7, Sep. 2006. [Online]. Available: http://doi.acm.org/10.1145/1163514.1163533

[3] P. Alexander, "Component-based software development", pp. 1834–1837 vol.3, 2001.

[4] P. G. Armour, "Beware of counting loc", Commun. ACM, vol. 47, no. 3,pp. 21–24, 2004.

[5] M. H. Halstead, "Elements of software science", Elsevierence, 1977.

[6] T. J. Mccabe, "A complexity measure", IEEE Transactions on Software Engineering, vol. 2, no. 4, pp. 308–320, 1976.

[7] J. Chen, H. Wang, Y. Zhou, and S. D. Bruda, "Complexity metrics for component-based software systems", International Journal of Digital Content Technology Its Applications, vol. 5, no. 3, pp. 235–244, 2011.

[8] S. Yu and S. Zhou, "A survey on metric of software complexity", in The IEEE International Conference on Information Management and Engineering, 2010, pp. 352–356.

[9] U. Kumari and S. Upadhyaya, "An interface complexity measure for component-based software systems", International Journal of Computer Applications, vol. 36, no. 1, pp. 46–52, 2011.

[10] H. Zuse, Software Complexity: Measures and Methods. Walter de Gruyter & Co., 1990.

[11] J. Shao and Y. Wang, "A new measure of software complexity based on cognitive weights", Journal of Electrical Computer Engineering Canadian, vol. 28, no. 2, pp. 69–74, 2003.

[12] S. Misra, "Modified cognitive complexity measure", in Computer and Information Sciences – ISCIS 2006, A. Levi, E. Savaş, H. Yenigün, S. Balcısoy, and Y. Saygın Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1050–1059.

[13] A. P. Singh and P. Tomar, "The analysis of software metrics for design complexity and its impact on reusability", in 2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom), March 2016, pp. 3808–3812.