Journal of Computers Vol. 31 No. 5, 2020, pp. 310-323
doi:10.3966/199115992020103105024

Syntax and Operational Semantics of a Component and Its Application in Maze Software

Ming Gu*

Department of Artificial Intelligence, ShenZhen Polytechnic, Shenzhen, Guangdong, China
gum@szpt.edu.cn

Received 1 August 2020; Revised 1 September 2020; Accepted 23 September 2020

Abstract. The guiding ideology of CBSE is to integrate and assemble the required components to generate the final required application software. In the detailed design stage of the software, due to the lack of formal component design methods, it is difficult to directly integrate the results of the detailed design stage into application software automatically. In this paper, formal rules BNF and SOS (Structured Operational Semantics) are used to describe the syntax and semantics of components at this stage, and nested references are used to describe the reference semantics among components. Taking the detailed design phase of the maze software as an example, it illustrates the specific application of the formal syntax and semantics proposed in this paper between the recursive components of the maze and other components of the maze software. The simulation survey comparison statistics with the two commonly used design tools at this stage prove that the component method of the detailed design stage proposed in this paper has the highest aggregate average score of comprehensive indicators.

Keywords: application software, component, maze recursive, semantics, syntax

1 Introduction

In software engineering, the software life cycle divided by time includes problem definition and feasibility analysis, requirement analysis, general design, detailed design, coding, debugging and testing, acceptance and running, maintenance and upgrading or obsoleting stages. With the development of software development technology, CBSE (Component-Based Software Development) have become more mature [1]. Generally speaking, CBSE is to integrate the required components into the final required application software [2]. Therefore, the component becomes the basic unites for developing application software. According to different stages of the software life cycle, the granularity of a components can be large or small one, the large one can be requirements documents, tools, files, directories and so on, and the small one can only have a piece of programming codes.

According to the main key idea of CBSE, the development of application software is a process of construction and evolution, and construction and evolution have become the two most basic characteristics of software systems. Construction is embodied in the use of components to integrate and assemble application software. According to the needs of the software life cycle and putting components of various granularities into the component base, the corresponding components in the component base are used and the integration and assembly task is completed at each stage of the software life cycle

In CBSE, formalization and software development methods are combined, making the research of formal software development methods a hot spot [3-4]. The research includes the consistency of application software evolution [7], the use of category theory to study component formal semantic description and the combination of components [8], the research on component retrieval in component libraries from the perspective of syntax and semantics [9], and using SOS to analyze and interpret the while loop structure in programming language [10] etc.. From the component model, the 3C model [11] is a generally recognized model, consisting of three parts: concept, content, and context. This model has a

* Corresponding Author

certain macroscopic guiding significance. However, there is no specific description of the syntax and semantics of the component. The REBOOT model [12] is a facet classification and retrieval model based on existing software components. It uses the term combination of a finite-dimensional information space to describe a component from the comprehensive perspective of several facets, and divides the description term space of the component into different facets which are conducive to the query and retrieval of components. The three mainstream industry standard component models are CORBA/OM of OMG, OLE/DCOM of MICROSOFT and Enterprise JavaBean of JAVA. There are also RESOLVE model and Beiing University Jade Bird component model [13] and so on. The key technology of these models provides componentized processing of various functional modules and functional reuse on this basis.

It can be seen from the above model that the research focuses on functional reuse, and the model focuses on the description of component functions and the interaction of internal and external interfaces related to component function interfaces. But to make software development truly embark on the road of engineering and industrialization, formalization and standardization are extremely important, and standardization is inseparable from formalization, which is one of the basic requirements of standardization.

From the current research situation around CBSE, there is no lack of research on formal technology, but there is a lack of formal and standardized component languages that describe components in each stage of the software life cycle from a formal perspective. In the software detailed design stage, there is also a lack of research on the formal description of the syntax and semantics of component description languages at the same time.

In the detailed design stage of the software, currently commonly used description methods include program flowcharts, NS diagrams and pseudocodes and so on. The description methods of various graphics should not be too complicated in structure. Only the general framework of programming algorithms can be listed. Some algorithm details are very hard to get involved. Pseudocode is more suitable for getting started, and the algorithm can also be described in more detail. But the disadvantage is that there is no way to look at the overall structure and the relationship between members from the macro delivery. The description is relatively random and the description results are not standardized. The common shortcoming of the commonly used methods at this stage is that there is no formal syntax and semantics. They must be converted to a specific programming language, such as C, Java, and Python, before they can be translated by different compilers and interpreters. The main reason is that these are not formal means and tools, and automatic conversion cannot be done directly with a language compiler or interpreter, and it is difficult to complete the goal of automatic integration and assembly to generate application software required by software construction in CBSE.

The research motive and purpose of this paper are to describe the detailed design stage of the software with the component method, and form a formal component description language at the same time for the formal description of the syntax and semantics of the component [5-6]. On the one hand, the purpose of the research is to provide a formal description method of the detailed design stage of software, and provide formal and automated support for the automatic integration and assembly of software. On the other hand, in the detailed design stage of software, a component description language is formed by describing the syntax and semantics of components. For component description languages, a compiler or interpreter of the component description language similar to the current various programming language compilers or interpreters will expect to be developed. Then, with the help of component compiler or interpreter and component base, the goal of automatic integration and assembly to generate application software required by software construction in CBSE is completed. Component has formal syntax and semantic description, which is the basis for designing component description language and developing component compiler or interpreter.

The main contributions of this paper are summarized as follows.

- The component proposed in this paper provides a formal description method in the detailed design stage of the software life cycle.
- The description of component syntax and semantics proposed in this paper provides a formal research ideas and support for component formalization and standardization.
- A component described with formal syntax and semantics is beneficial to the development of component compiler or interpreter.

This paper is divided into six chapters: The first chapter is a brief introduction, the basic introduction

to problem of other existing related research work, the research motive and the main contributions in this paper; The second chapter describes the syntax of a component using BNF in the detailed design stage of software; The third chapter describes the SOS (Structured Operational Semantics) of a component and the reference semantics among components in the same stage of software; The fourth chapter is an application of the proposed syntax and SOS in maze software; The fifth chapter is a simulation survey comparative statistics; The last chapter is a conclusion and further prospect.

2 BNF (Backus- Naur Form) Description of a Component in Detail Design Stage

The syntax rules of the component are composed of eight parts, which are control range (may be empty), returned value (may be empty), component name, parameter list (may be empty), component base name of the component, attribute list, relation list and operation list in order.

The eight parts are described using BNF as follows.

<Comp_spec> ::= COMPONENT [<Con_area>]
[<Return_type>]
<Comp_name>
[(<Comp_para_list>)]
(<Base_fields>)
(<Attribute_list>)
(<Relation_list>)
(<Operation_list>) END

It can be seen from the above that the component can contain up to eight parts grammatically, starting with the keyword COMPONENT and ending with the keyword END. The content in the square brackets can be empty. These eight parts will be described in detail one by one below.

2.1 Syntax of the Component Control Range

The syntax of the component control range can be further described as follows. Public, Protected, Default and Private are key words.

<Con_area> ::= Public | Protected | Default | Private

It explains the access control range of the component when it is used, that is, within what range it can be used. Because it is a component in the detailed design stage, it is equivalent to the access control authority of the class which is used in OOP Java language. Among them, public means it can be used in any situation, protected means it can be used in a specific application software structure, default means it can be used in the component base of the same name, private means it can be used in the same component base. When the application software is constructed, private component can be used by other components in the same component base. The component control range may be empty. For example, it's not needed when the C language is chosen in programming coding.

2.2 Syntax of Component Returned Value

The syntax of the component return value can be further described as follows.

<Return_type> ::= String | Int | Float | Double | Char | Array | Struct | Self_def | Void

The String | Int | Float | Double | Char is a data type in general OOP or process-oriented (such as C language) programming. The concept of programming language is used here. The Array and Struct represents the set type of data, such as Arrays and structures in C language, the Arraylist class in the Java system package or user-defined Java array, and the combined data types in Python language, etc. The Self_def means that there can be user-defined types, such as typedef in C language and so on. The Void means that there is no return value. Component returned value may be empty.

2.3 Syntax of the Component Name

The syntax of the component name can be further described as follows.

<Comp_name> ::= Σ^*<Digits> | Σ^*
Σ ::= { a–z | A–Z | _ | $ }
$\Sigma^0 = \{ \varepsilon \}$ ($x\varepsilon = \varepsilon x, x \in \Sigma$)
$\Sigma^n = \Sigma\Sigma^{n-1}$ ($n \geq 1$)
$\Sigma^* = U_{n\geq 0} \Sigma^n$
<Digits> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

The component name is an identifier, which must begin with a letter or underscore (_) or dollar sign ($) and may contain only letters, underscores, dollar signs, or digits. The component name of a component cannot be empty, that is, the component must have an identifier component name. In the entire application software, the component name in the same component base must be unique.

2.4 Syntax of the Parameter List

The syntax of the parameter list can be further described as follows.

<Comp_para_list> ::= <Return_type> <Comp_para,>*
<Comp_para> ::= Σ^* <Digits> | Σ^*

The parameter list is composed of one or more parameters. If the parameter list is not empty, it contains at least one parameter. Each parameter is composed of two parts, one part is <Return_type> which has been defined in the syntax of the component returned value that is a recursive definition and the other one is component identifier. <Return_type>follows a space. The parameter in parameter list is a formal parameter. When the component is actually used, the formal parameters are replaced with actual parameters. The parameter table of a component can be empty.

2.5 Syntax of the Name of the Component Base

The syntax of the name of the component base can be further described as follows.

<Base_fields> ::= <Base_name>*
<Base_name> ::= Σ^* <Digits> | Σ^*

The component base name provides a means to organize related components together according to certain characteristics. This characteristic can be an application domain or different stages of the software life cycle etc. A component can also belong to multiple different component libraries. It can provide convenience for practical applications although Information redundancy of component exists.

2.6 Syntax of the Attribute List

The syntax of the name of the Attribute list can be further described as follows.

<Attribute_list> ::= (<Inside_attr>)(<Env_clause>)
<Inside_attr> ::= <Prim_attr>*[CT:<Create_time>,][MT:<Modify_time>,][UF:<Use_frequency>]
<Prim_attr> ::= <Return_type>
<Create_time> ::= Year-Month-Date

Using 4 digits to indicate the year, 2 digits to indicate the month and day, and separate the year, month and day with “-”.

<Modify_time> ::= Year-Month-Date
<Use_frequency> ::= <Digits>*

<Env_clause> ::= <Version_spec>,<Develop_env>,<Run_env>,<Documetation_name>
<Version_spec> ::= V <Digits>*. <Digits>*
<Develop_env> ::= S: <Source>[;L:<Lib>];C:<Compiler>[;D:<Database>][;N:<Net>];O:<OS>
<Source> ::= <Language_name>

The <Language_name> is the programming language used to implement the component, such as: C, Java, Python, etc.

<Lib> ::= <Library_name>

The <Library_name> is the programming language library, package, or third-party library or package used to implement the component.

<Compiler> ::= <Compiler_name>

The <Compiler_name> is the compilation environment used by the component, such as: Visual Studio2019, Eclipse, Pycharm, etc.

<Database> ::= <Database_name>

The <Database_name> is the database name used by the component, such as: SQL Server, Access and Oracle etc.

<Net> ::= <Net_name>

The <Net_name> is the network environment resources used by the component, such as Alibaba Cloud, OpenStack, etc.

<OS> ::= <OperatingSystem_name>

The <OperatingSystem_name> is the operating system support used by the component, such as: Windows, Linux, etc.

<Run_env> ::= <O:OS>[;C:<Compiler>][;<D:Database>[;<N:net>]
<Documention_name> ::= F:<File_name;>*

The <File_name> is the document used by the component, expressed in the form of file name. There can be multiple file names, and the path name can be preceded by the file name.

Attributes are composed of internal attributes and environmental attributes. The internal attributes include some predefined primitive types, such as integers, real numbers, characters, Booleans, strings, arrays, and structures, which are defined recursively with <Return_type>. Internal attributes include creation time, modification time and frequency of use.

Environment attributes include version description, component development environment, component running environment and document name related to the component.

2.7 Syntax of the Operation List

The syntax of the operation list can be further described as follows.

<Operation_list> ::= <Statements>*
<Statements> ::= <Assignments> <Selections> <Repetitions>
<Assignments> ::= <Variable>=<Arithmetic_expression>
<Arithmetic_expression> ::= <Variable> | <Digits> | (<Arithmetic_expression>
<Arithmetic operator> <Arithmetic_expression>)
<Arithmetic operator> ::= + | -| * | / | %

<Variable> ::= Σ^*
<Selections> ::= if <Boolean_statement> <Statements> else <Statements> | Switch (Int)
<Case_statements>* [default < Statements>*]
<Case_statements> ::= case Int <Statements> * [break]
<Boolean_statement> ::= <Arithmetic_expression> <Relational_expression> <Arithmetic_expression>
<Relational_expression> ::= < | > | <= | >= | == | ! = | And | Or | Not
<Repetitionis> ::= while <Boolean_statement> <Statements>* | do <Statements> * while <Boolean_statement> |
for ([<Statements>*]; [<Boolean_statement>]; [<Statements>*]) <Statements>*

The operation list is a collection of statements. In the detailed design stage of the statement, In accordance with the principles of structured programming and statements expressed in pseudo code, there are three forms of assignment, condition, and loop.

2.8 Syntax of Relation List

The syntax of the relation list can be further described as follows.

<Relation_list> ::= I:<Invoking_R;>* | [,C:<Called_R;>*]
<Invoking_R> ::= (<Base_fields>. <Comp_name>)*

The <Invoking_R> Indicates which components in a component base are called by the component. This component is the calling one, and the component represented by <Invoking_R> is called.

<Called_R> ::= (<Base_fields>. <Comp_name>)*

The <Called_R> Indicates which components are called by the component. This component is called, and the component represented by <Called_R> is the calling one

There are two relations between components: calling relationship and called relationship. The called relationship may be empty, which means that the component is the startup component of the application software. If the calling component and the called component are regarded as two sets, then <Invoking_R> ∩ <Called_R> ≠ Φ(Φ means empty) is true. For example: recursive call means that the intersection is not empty.

3 SOS of a Component

Formal semantics is an integral part of programming theory. It uses mathematics as a tool and uses symbols and formulas to rigorously define the semantics of programming languages and formalize them. Operational semantics is a kind of formal semantics. Operational semantics describes the meaning of programming language by specifying the execution process of programming language on abstract machines, that is, the semantics of language statements are their corresponding computer operations. Plotkin proposed a method of structured operational semantics in 1981. The basic idea is that the operational semantics of compound statements should be reduced to the operational semantics of its various components. Its distinctive feature is that the program relies on a series of explicit commands for state transition when the program is running. SOS is described by a set of rules which stipulate how expressions are calculated and how commands are executed.

SOS consists of three parts. The first is syntax categories, such as variables, constants and function identifiers etc. The second is syntax rules called static semantics in which all legal statements structure is given. The basic rules are given in the form of axioms and the auxiliary rules are given in the form of inference in the second part. The third is dynamic semantics given in the form of converted triples, specifically describing the changes in the state after executing a syntax component.

We have used BNF to describe the syntax of the component. After giving a few related definitions, the SOS of the component is described.

3.1 Related Definitions

Definition 1: The s is used to indicate that s is a legal language element. The $\frac{s}{t}$ indicates that if the s is a legal language element, the t is also a legal language element.

Definition 2: If $s\&t$ is a legal language element, $s;\ t$ is also a legal language element. The previous sentence is expressed as $\frac{s\ \&\ t}{s;t}$.

Definition 3: The dynamic semantics is composed of a set of rules. The rules describe the state changes after executing a language element. The basic element of the rules is the Combination status, which is represented by $<s,\ \sigma>$. The meaning of the $<s,\ \sigma>$ is current state is σ and the statement segment to be executed is s. The semantics of executing an assignment statement can be described as $<x:=e,\ \sigma> \rightarrow <\sigma[e/x]>$. It indicates that after the statement $x:=e$ is executed, the original state σ has changed in which the original value of x is replaced by the value of expression e.

Definition 4: The rule reasoning form is $\frac{<s,\sigma>\rightarrow<s',\sigma'>}{<s;t\ \sigma>\rightarrow<s';t,\sigma'>}$. It means that if the result of running the program is the s changes into the s' and the state σ changes into the state σ', the result of running the program is the $s;\ t$ changes into s'; t and the state also changes from σ to σ'.

Definition 5: An event is the trigger source of component operation and an action that occurs instantaneously. The event may be a function call, a class-generated object, a third-party library, or a message, instruction, etc.. The set of events is represented by $E=\{e_n \mid n\geq 0\}$.

Definition 6: The component state refers to the combination of the four-tuple <parameter table, attribute table, relationship table, operation table> of the component at a given moment, marked as SC = < [Para], Attr, Rela, Oper>. According to the syntax of the component, Para may be empty. When any one element in the four-tuple has changed, it is said that the state of the component has changed. The change of component state is triggered by events. If all the instruction sequences of the operation list of the component SC have been executed normally, it is called the normal end state of the component, which is represented by Norm. If the operation list of the component is not completely executed, the abnormally terminated state is called the abnormal state of the component, which is represented by Abort. If a component enters a state that cannot be terminated, it is called the infinite loop state of the component, which is represented by Endless.

For a given component state, component events can be sorted in chronological order. The expression $e_i <e_j$ is correct if and only if e_i occurs before e_j. At a given moment, only one event can occur. The expression $e_i = e_j$ is correct if and only if ei and ej are the same event.

Definition 7: The life cycle of a component is the set of all component state changes, which constitutes the life cycle of the component. Because the basic cause of state changes is events, the life cycle of a component can be defined as a Cartesian product of the state of the component and the event that triggers all component state changes. It is expressed as CL=SC×E.

3.2 SOS Description of the Component

(1) Syntax Category

The syntax category includes the following five parts:

Variable Set: $V=\{v_i \mid i = 0, 1, \ldots, n\}$

Arithmetic Expression Set: $A=\{a_i \mid i== 0, 1, \ldots, n\}$

Relational Expression Set: $R=\{\text{true, false}, r_i \mid i== 0, 1, \ldots, n\}$

Operation Set (Statement Set): $S=\{s_i \mid i== 0, 1, \ldots, n\}$

The s_i means assignment, selection or repetition statements.

(2) Static Semantic (Syntax rules)

Assignment statement itself is a legal language element. The formula is as follows.

$$v_i:=a_i \mid r_i. \tag{1}$$

If the operation set s_1 and s_2 are legal language elements, the "if " statement in the operation list defined by the following component syntax is also a legal language element. The syntax rule is as follows.

$$\frac{s_1 \,\&\, s_2}{\text{if } r_1 \; s_1 \text{ else } s_2}. \tag{2}$$

$$\frac{s_1 \,\&\, s_2}{\text{switch (Int) } s_1; s_2}. \tag{3}$$

If the operation set s_1 *and* s_2 are legal language elements, the "switch*"* statement in the operation list defined by the following component syntax is also a legal language element. The syntax rule is as follows.

If the operation set s is legal language elements, the "while" statement in the operation list defined by the following component syntax is also a legal language element. The syntax rule is as follows.

$$\frac{s}{\text{while } r_i \; s}. \tag{4}$$

If the operation set s is legal language elements, the "do-while" statement in the operation list defined by the following component syntax is also a legal language element. The syntax rule is as follows.

$$\frac{s}{\text{do } s \text{ while } r_i}. \tag{5}$$

If the operation sets s_1 and s_2 are legal language elements, the sequentially executing statements s_1; s_2 defined by following component syntax of operation list is also a legal language element. The syntax rule is as follows.

$$\frac{s_1 \,\&\, s_2}{s_1; s_2}. \tag{6}$$

If the operation set s_1, s_2, s_3, and r_i are legal language elements, the "for" statement in the operation list defined by the following component syntax is also a legal language element. The syntax rule is as follows.

$$\frac{s_1 \,\&\, s_2 \,\&\, s_3 \,\&\, r_i}{\text{for } s_1; r_1; s_2 \; s_3}. \tag{7}$$

(3) Dynamic Semantic (Transformational rule)

The current state of the component is SC. If the assignment statement $v_i{:=}a_i$ is executed, the original state SC will change and the value of variable v_i is replaced by the value of expression a_i. Because SC is a four-tuple, this substitution occurs in every part of SC. The transformational rule is as follows.

$$< v_i{:=}a_i, \text{SC}> \rightarrow <\text{SC}[a_i / v_i]>. \tag{8}$$

If eval $<r_i$, SC> = true is a rule, the execution result of the "if " statement is to execute s_1 and change the component state from SC to SC'. The transformational rule is as follows.

$$\frac{\text{eval} < r_i, \text{SC} >= \text{true}}{< \text{if } r_i \; s_1 \text{ else } s_2, \text{SC} > \rightarrow < s_1, SC' >}. \tag{9}$$

If eval $<r_i$, SC> = false is a rule, the execution result of the "if " statement is to execute s_2 and change the component state from SC to SC'. The transformational rule is as follows.

$$\frac{\text{eval} < r_i, \text{SC} >= \text{false}}{< \text{if } r_i \; s_1 \text{ else } s_2, \text{SC} > \rightarrow < s_2, SC' >}. \tag{10}$$

If eval $<a_i$, SC> = a_j is a rule, the execution result of the "switch" statement is to execute s_j and change the component state from SC to SC'. The transformational rule is as follows.

$$\frac{\text{eval} < a_i, \text{SC} >= a_j}{< \text{switch (Int) } s_j; s_k, \text{SC} > \rightarrow < s_j, SC' >}. \tag{11}$$

If eval $<r_i$, SC> = true is a rule, the execution result of the "while" statement is to execute while again

after executing *s* and change the component state from SC to SC'. The transformational rule is as follows.

$$\frac{\text{eval} < r_i, \text{SC} >= \text{true}}{< \text{while } r_i\ s, \text{SC} > \rightarrow < s; \text{while } r_i\ s, \text{SC}' >}. \tag{12}$$

If eval <r_i, SC> = false is a rule, the execution result of the "while" statement is that *s* is not executed, the while statement is terminated and the component state SC does not change. The transformational rule is as follows.

$$\frac{\text{eval} < r_i, \text{SC} >= \text{false}}{< \text{while } r_i\ s, \text{SC} > \rightarrow < \text{SC} >}. \tag{13}$$

If eval <r_i, SC> = true is a rule, the execution result of the "do-while" statement is to execute do-while again after executing *s* and change the component state from SC to SC'. The transformational rule is as follows.

$$\frac{\text{eval} < r_i, \text{SC} >= \text{true}}{< \text{do } s \text{ while } r_i, \text{SC} > \rightarrow < s; \text{do } s \text{ while } r_i, \text{SC}' >}. \tag{14}$$

If eval <r_i, SC> = false is a rule, the execution result of the "do-while" statement is that do-while not to execute after executing *s* and change the component state from SC to SC'. The transformational rule is as follows.

$$\frac{\text{eval} < r_i, \text{SC} >= \text{false}}{< \text{do } s \text{ while } r_i, \text{SC} > \rightarrow < s, \text{SC}' >}. \tag{15}$$

If eval <r_i, SC> = true is a rule, the execution result of the "for" statement is that s_1 is executed first, s_3 and s_2 are executed sequentially. The s_1 will not be executed after the first time. The state of the component changes from SC to SC'. The transformational rule is as follows.

$$\frac{\text{eval} < r_i, \text{SC} >= \text{true}}{< \text{for } s_1, r_i, s_2; s_3, \text{SC} > \rightarrow < s_1, s_3, s_2; \text{for } r_i, s_2, s_3; \text{SC}' >}. \tag{16}$$

If eval <r_i, SC> = false is a rule, the execution result of the "for" statement is that s_1 is executed, s_2 and s_3 are no longer executed. The component state changes from SC to SC'. The transformational rule is as follows.

$$\frac{\text{eval} < r_i, \text{SC} >= \text{false}}{< \text{for } s_1, r_i, s_2; s_3, \text{SC} > \rightarrow < s_1, \text{SC}' >}. \tag{17}$$

If the execution of the statement set *S* makes the component state reach the normal final state (Norm) in the component state SC, the statement set *S* and the statement s_j are executed in sequence means that there is a statement s_j are waiting to be executed under the normal final state of the component. The transformational rule is as follows.

$$\frac{< S, \text{SC} > = < \text{Norm} >}{< S; s_j, \text{SC} > \rightarrow < s_j, \text{Norm} >}. \tag{18}$$

In the component state SC, if the execution of the statement set *S* makes the component state reach the abnormal state, the statement set *S* and the statement s_j are executed in sequence means that the statement set *S*' which is the subset of *S*, that is $S' \subset S$, and the statement s_j to be executed in the abnormal state of the component are no longer having any meaning. From the semantics of program execution, the program terminates abnormally when the operation set has not ended in sequence. The transformational rule is as follows.

$$\frac{< S, \text{SC} > = < \text{S}', \text{Abort} >}{< S; s_j, \text{SC} > \rightarrow < s_j, \text{Abort} >}. \tag{19}$$

In the component state SC, if the execution of the statement set *S* makes the component reach the

infinite loop state (Endless), the statement set S and the statement s_j are execute executed sequentially means that in the infinite loop state, there will always be a statement set S' and Statement s_j. There is $S' \subset S$, that is, S' is a proper subset of S. From the semantics of program execution, the operation list can never be completed until the system resources are exhausted. The transformational rule is as follows.

$$\frac{< S, \mathrm{SC} > = < \mathrm{S}', \mathrm{Endless} >}{< S; s_j, \mathrm{SC} > \rightarrow < S'; s_j, \mathrm{Endless} >}. \tag{20}$$

3.3 Description of Reference Semantics Among Components

Reference is a kind of temporary association among components. For example, function call in C language is a kind of reference. The inheritance between parent and child classes in Java language and the realization of class-to-interface can also be regarded as a kind of reference. The usage of third-party libraries is also a reference in Python language.

If component A references component B, or B is referenced by A, we call A as the referencing component and B as the referenced component. The component life cycle of A is CL(A) and the B is CL(B).

According to definition 7, CL=SC×E, the Cartesian set is regarded as a set, then the reference semantics between components can be described as: CL(B) $\subset$ CL(A), the intuitive meaning of this description is that it is the life cycle of the referenced component is contained in the life cycle of the referencing component. It means that the life cycle of the referencing component is longer than the life cycle of the referenced component.

For example, in C language, after the called function returns control or data to the caller, the calling function processes the control and data returned by called function. The life cycle of the calling function is longer than the life cycle of the called function. In Java language, the parent class must have been created before the subclasses are created. In Python, the usage of third-party libraries is similar to functional calling in C language.

Definition 8: The reference among component can be nested. It means that the life cycle of the component can also be nested. Assume that the life cycle set of the component is CL=$\{C_i \mid i=1, ..., n\}$, $\forall C_i, C_j, C_k, C_n \in$ CL, $\exists C_i \subset C_j \subset C_k \subset \ldots \subset C_n$, it means that C_i refers to C_j, C_j refers to C_k, C_k can refer to other components, and finally C_n can also be referred.

Nested reference among components that meet certain syntax and semantics specifications can be integrated and assembled into the final application software.

4 Application of Syntax and SOS Description in Maze Software

The core of maze software is a recursive component. Recursion means that a component refers to itself, and the parameter list is different every time it refers. Inside the recursive component, there must be a statement to determine whether the component is terminated, otherwise the process of component reference will enter the Endless or Abort state.

According to the syntax and semantics of components, the design of recursive components is to do the two important things. The first one is to design of the component parameter list prevents the component from reaching the Abort and Endless states. Another one is to design the exit condition of the operation list in the component is completely designed so that the component can reach the Norm state.

We use component syntax and semantics to describe the more complex maze recursive component. The running results of integrated and assembled into the final application software including the maze recursive component are given. The final application software is implemented by using C programming language.

4.1 Syntax Description of Maze Recursive Component

According to BNF description of a component, the maze recursive component is described as follows.

(1) Control range is default. Returned value is void. Component name is maze. Parameter list is (struct **maze, int dimension, int row, column, char* path). The first four parts of BNF are as follows.

COMPONENT default void maze (struct **maze, int dimension, int row, column, char* path)

(2) We assume that the component base name of the maze component is CLanguage_base. The fifth part of BNF is as follows.

(CLanguage_base)

(3) According to the syntax of the component attribute list, the internal and environmental attributes of the maze component are described in sequence. The sixth part of BNF is as follows.

(int, char, CT:2020-3-1, MT:2020-7-30, UF:3) (V1.2, S: C_language; L: head.h; C: Visual_studio_2019; O: Windows2010, O: Windows2010, C: Visual_studio_2019, F:D:\CLanguage\maze.c; F:D:\CLanguage\mazeHead.h)

(4) According to the syntax of component operation list, maze component operations are described in sequence. The seventh part of BNF is as follows.

```
(if (Encounter various boundaries) return
if (Encounter various obstacles) return
if (Encountered the middle point of the maze)
   if (Encounter exit)
   String processing assignment statements
    print a successful path
   else
    mazePaths (struct **maze, dimension, row, column+1, new_path)
    mazePaths (struct **maze, dimension, row+1, column, new_path)
    mazePaths (struct **maze, dimension, row, column-1, new_path)
    mazePaths (struct **maze, dimension, row-1, column, new_path))
```

(5) According to the syntax of the component relation list, the relation of the maze components described in sequence. The eighth part of BNF is as follows.

(I:CLanguage_base. print_ paths, C: CLanguage_base. main) END.

4.2 The SOS and Reference Semantics of Maze Recursive Component

(1) The SOS of Maze Recursive Component

The operation list of the maze recursive component uses assignment statements and if statements, static semantics (syntax rules) involve formula (**1**), (**2**) and (**6**), dynamic semantics (transformational rules) involve formula (**7**), (**8**), (**9**), (**10**) and (**18**). Formula (**18**) describes the maze recursive component reaches the Norm state. If the "if " statement or parameter list of the maze recursive component is not properly designed, the dynamic semantics (transformational rules) will involve formula (**19**) and (**20**).

The unreasonable design of the "if " statement of the maze recursive component includes that the exit situation is not well considered, it may involve formula (**19**) and (**20**). The properly design of the parameter list includes that there are four walking directions at any intermediate point of the maze, that is, the front, back, left, and right, the coordinate value of the row and column should consider each walking direction, otherwise it will lead to the occurrence of formula (**20**).

(2) The Reference Semantics of Maze Recursive Component

The reference semantics of the maze recursive component, according to the description of the maze recursive component relation list above, maze recursive component called component is print_paths, and the calling component is main. Assume that the life cycles of these three components are CL (maze), CL (print_paths), CL (main), according to Definition 8, the semantics of their nested reference relation is: CL (print_paths) $\subset$ CL (maze) $\subset$ CL (main)

4.3 Integrated Assembly of Maze Application Software

(1) Brief Requirement Analysis and General Design of Maze Application Software

A section of path in maze is represented by a dot, and the number on the dot represents the cost of passing through this path. We represent the cost from 0 to 9. The larger the number, the more cost. The minimum cost is 0 and the maximum cost is 9. The asterisk (*) means a wall or an obstacle which indicates that the path cannot be passed through. The maze is a square, represented by a mathematical model as an (n*n) square matrix, which is composed of numbers and *. Given a starting point, if you can find a string of numbers from the starting point to a certain number in the rightmost column, there is a path. Different digital strings form different paths. When counting the number of passages in the maze, only the number of different number strings needs to be counted. The shortest path means that the number of digits in the digital string is the least, the cheapest path means that the sum of digits in the digital string of a certain path is the smallest, and the total cost of the cheapest path is the sum of the numbers on the cheapest path. For example, $\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 0 & 7 \\ * & 8 & 9 & * \\ 2 & 4 & * & 6 \end{bmatrix}$ represents a square maze consisting of 4*4, if the starting point specified is 1 which is in the upper left corner. The exit is on the rightmost column. There are 3 exit points. The numbers on the exit points are 4, 7, and 6 respectively.

(2) Detail Design and Coding of Maze Application Software

After describing the maze recursive component with its syntax and semantics, we integrated and assembled maze application software with the maze recursive component as the core, and the application software was programmed in C language.

In maze application software, the user is asked to enter the coordinates of the entrance row and column of the maze. The application software uses the recursive call of the maze component to print the number of paths in the maze, the shortest path, the cheapest path and the total cost of the cheapest path. Maze application software has the following 8 components. They are main (startup), maze (recursive), print_paths (print a maze path), get_maze_dimension (read maze dimensions from disk file), parse_maze (read maze matrix from disk file), display_shortest_path (print shortest path), display_cheapest_path (print cheapest path), path_cost (print the total cost of the cheapest path).

The syntax and semantics of the core maze recursive components have been described above. In order to reflect the constructiveness of the application software, here is only the semantics of the nested reference relation among components of the maze application software. They are

CL (print_paths) $\subset$ CL (maze) $\subset$ CL(main), CL (get_maze_dimension) $\subset$ CL (main)
CL (parse_maze) $\subset$ CL (main), CL (display_shortest_path) $\subset$ CL (main)
CL (display_cheapest_path) $\subset$ CL(main), CL (print_paths) $\subset$ CL (display_cheapest_path)
CL (path_cost) $\subset$ CL (display_cheapest_path), CL (print_paths) $\subset$ CL (display_shortest_path)

The following Fig. 1 is a screenshot of the running interface of the application software containing the maze recursive component implemented in C language.

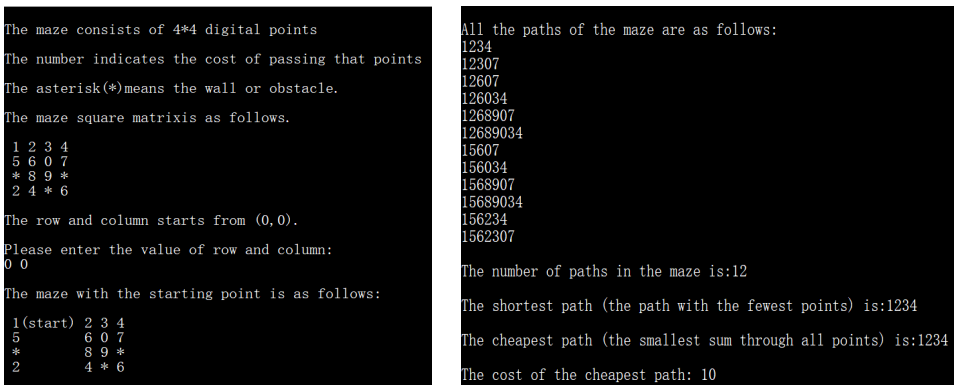

Fig. 1. Screenshot of the running interface implemented in C language

5 Simulation Survey Comparative Statistics

In the detailed design stage of the software, for the two commonly used description tools which are program flow chart and pseudocode, in order to compare with the component method proposed in this article, we have designed some statistical indicators for simulation comparison, and the statistical results obtained through website data collection are shown in Table 1.

Table 1. Data comparison statistics from website survey

Survey statistical indicators		Program flow chart	Pseudocode	Component in this paper
Arithmetic description	Adaptability	100	100	100
	Level of detail	80	100	100
Recognition degree of automatic compilation and interpretation software		50	80	90
Syntax		50	50	90
Semantics		50	50	90
Amount of Information	Internal and external attributes	80	90	90
	Mutual relations	50	80	90
	Network and database	80	80	90
	Application expectations	80	80	50
Concise and intuitive		100	80	50
Aggregate average		72	79	84

In simulation survey statistics, the basis for designing survey statistics indicators is to first consider the role and purpose of the evaluation object, and also refer to the software quality evaluation standard of the reference document [14]. There are six categories and ten subcategories. The collection of statistical data is obtained through the aggregation of information from the website and various groups. The data in Table 1 is the average value of various types of information. The full score of each indicator is 100, and there are five grades of 0,50,80,90,100 for every statistics indicator.

The program flow chart and pseudocodes are often used, and statistical results can be collected directly from website. For the component methods mentioned in this paper, some indicators are self-evaluation, and some are feedback evaluations after a brief introduction sent to website, various groups and so on.

It can be seen from the survey statistics, from indicator of Concise and intuitive and Application expectations, according to the evaluation feedback of the indicators, the program flow chart is the most popular. When you want to describe the algorithm in the detailed design phase as a whole, and you do not need detailed details, just describe the algorithm framework intuitively, you will basically use the program flow chart. Therefore, the program flow chart has always been a favorite tool for software personnel and teaching. Except for syntax and semantic indicators, pseudocode is more popular, and all indicators are in the middle position, and the score of describing the algorithm is the same as the component method suggested in this article. The limitation of pseudocode is the results after description vary from person to person. Except for the Application Expectations, Concise and Intuitive, the other indicators of the component method proposed in this paper are better than or equal to the pseudocode method. Generally speaking, because people have a time delay in accepting new things, especially for abstract formal methods, this time delay will increase if useful compiler or interpreter tools cannot be provided effectively.

All in all, the comparison results after simulation survey and statistics show that the component method proposed in this paper has the highest aggregate average score. If the support of compiler or interpreter tools is provided, the component method proposed in this article will have good application expectations.

6 Conclusion

In CBSE, this paper explores a kind of software constructive research ideas and approaches. The formal component research in the detailed design stage of the software is proposed. First, the formal rule BNF is used to describe the syntax of the component at this stage, and then the formal SOS is used to describe the semantics of the component at this stage. The semantics of references among components are shown. Taking the maze software as an example, the maze recursive component is described. With the help of a C language compiler, the application software is integrated and assembled, and the application software for finding the maze path is realized in C language. The simulation survey and comparative statistics of commonly used detailed design stage description tools prove that the component method of detailed design stage proposed in this paper has the highest aggregate average statistical results.

In the future work, we will develop a compiler or interpreter that can recognize the syntax and semantics of the component described in this paper. There is no need to resort to compiler or interpreter in different programming languages such as C and so on. Through the component compiler or interpreter, the components described in BNF and SOS will be directly integrated and assembled into application software.

References

[1] Y.-H. Wang, Component Software Technology, Machinery Industry Press, Beijing, 2012 (Chapter 2-8).

[2] F.-Q. Yang, H. Mei, K.-Q. Li, Software reuse and software component technology, Acta Electronica Sinica 27(2)(1999) 68-75.

[3] X.-X. Ma, X.-Z. Liu, B. Xie, P. Yu, T. Zhang, L. Bu, X.-D. Li, Software development methods: review and outlook, Journal of Software 30(1)(2019) 3-21.

[4] J. Wang, N.-J. Zhan, X.-Y. Feng, Z.-M. Liu, Overview of formal methods, Journal of Software 30(1)(2019) 33-61.

[5] Y.-Q. Chen, Formal Languages and Automata Theory, Nankai University Press, Tianjin, 2001 (Chapter1-2).

[6] R.-Q. Lu, Formal Semantic of Computer Language, Science Press, Beijing, 1992 (Chapter 2).

[7] J.-J. Zhen, T. Li, Y. Lin, Z.-W. Xie, X.-F. Wang, L. Cheng, M. Liu, Judgement method of evolution consistency of component system, Computer Science 45(10)(2018) 189-195.

[8] S. Zhao, Research on formal description and combination of component, [dissertation] Zhejiang: Zhejiang Normal University, 2012.

[9] Q. Li, Research on formal description and retrieval of component based on ontology, [dissertation] Yunnan: Kunming: Kunming University of Science and Technology, 2017.

[10] X.-B. Ye, S. Wang, L. Zhi, Interpretation the while loop structure in programming language by using operational semantics, Journal of Chuxiong Normal University 25(12)(2010) 45-47.

[11] A. Strohmeier, S. Chachkov, A side-by-side comparison of exception handling in Ada and Java, Version 1.1, ACM Press, 2001.

[12] J. Morel, M.J. Faget, The REBOOT environment, Software Reusability, in: Proc. Advances in Software Reuse (1993) 80~88.

[13] http://www.360doc.com/content/10/0512/14/37945_27219603.html.

[14] Y.-L. Ye, S.-Y. Zhu, Software quality evaluation system and its implementation, Journal of Computer Applications and Software 18(1)(2001) 26-33.