A JAVA Code Generation Method based on XUML

Zhenyu Wang
Xi’an University of Posts and Telecommunications, Xi’an 710121, China; ruangongwwk@163.com

Abstract. In order to solve the problem that the semantic of UML is not precise enough to generate a complete program, an extensible language xUML is proposed in this paper. Firstly, the activity diagram of xUML is analyzed to obtain the Hierarchical Syntax Chart of the system, and then the JAVA program is obtained according to the Hierarchical Syntax Chart. The results show that the xUML language can generate a complete JAVA program.

1. Introduction
In today’s society, computer-related technology has gradually penetrated into all aspects of life. With the continuous emergence of new technologies, the development environment of application software has become more and more complicated. Under such circumstances, automatic code generation technology has gradually become a concern of scholars. The characteristic problem is that the source code is automatically generated according to the modeling of the user’s task requirements. Unified Modeling Language (UML) is the most widely used unified modeling language. It has the advantages of standard unification, modeling visualization, etc. However, its semantics are not precise enough to realize the complete conversion of the built model to the source code. Only the general framework of the model can be translated.

Many scholars at home and abroad have done research on automatic code generation. Albert M puts forward the design ideas and methods in the automatic code generation [1] of UML class diagrams. Pereira F introduces the conversion method [2] from Petri net to ANSIC language. Li Qingshan proposed a graph-based code generation algorithm [3] for reverse engineering.

In order to solve the problem of UML semantic inaccuracy, this paper proposes an extended Unified Modeling Language (xUML) based on UML [4], and designs a conversion algorithm from xUML activity diagram to JAVA program. Firstly, the formal definition of xUML data type, operator and activity diagram are proposed, then the conversion algorithm of xUML activity diagram to JAVA program is proposed, and finally the algorithm is verified by experiment.

2. Preliminary

2.1 The definition of xUML
According to the design goals and principles, the xUML language can establish both static and dynamic models of the system [5-6]. This article mainly describes the activity diagram in its dynamic model.

The dynamic model consists of a sequence diagram and an activity diagram. The sequence diagram is consistent with the UML sequence diagram. The activity diagram is an extension of the UML activity diagram. For example, the initial state adds the function name and parameters, the termination
state can contain the return value, and the function call contains the called function. The name and parameters of the activity call contain the name and parameters of the called activity.

2.1.1 Data types and operators

The common data types supported by xUML are shown in Table 1, and the operators are shown in Table 2:

| data types | Claim type |
|------------|------------|
| Boolean    | var1: Boolean          |
| Integer    | var2: Integer          |
| Real       | var3: Real             |
| String     | var4: String           |
| Set        | var5: Set (type)       |
| Sequence   | var6: Sequence (type)  |

In addition, xUML supports declaring one-dimensional or multi-dimensional variable arrays, such as arr[10]: Integer, which declares arr as an Integer array with a length of 10. The array can be globally assigned without subscripts.

The basic expression statements provided by xUML are shown as follows:

1) Declaration statements: including declaration variables, constants, objects, and functions, such as var: Integer, declaration var is an Integer type variable, and func(): Integer, which declares a func() function of type Integer;

2) Expression statement: consists of basic arithmetic operators, such as b * (c - d);

2.1.2 Dynamic Model Definition

Table 3 shows the element activity diagram in xUML:

Table 3. The element of activity diagram in xUML

| Element      | Name     | Description                                                                 |
|--------------|----------|-----------------------------------------------------------------------------|
| fun (param: type) | InitialState | The entry state of the dynamic model, including the function name and parameters. |
| [return]     | FinalState | The termination of a segment of an active process in a dynamic model can include a return value. |
|              | ActionNode | Actions are special cases of active nodes, and internal operations are atomic operations that can be performed. |
|              | ActionNode | It is an embedded action group and is the organizational unit of the activity. |
2.2 Formal definition of xUML activity diagram

In xUML, the activity graph can be viewed as a directed graph, where the node symbols (symbols other than the streamlines) are vertices, and the streamlines are arcs between vertices. Therefore, the xUML system model is actually a collection of directed graphs. Its formal definition is as follows:

\[ \text{ADModel} = \text{ADSet}, \text{entryAD} \]

\[ \text{AD} = \text{verSet}, \text{arcSet}, \text{entryVer} \]

\[ \text{Arc} = \text{verFrom}, \text{label}, \text{verTo} \]

\[ \text{Vertex} = \text{type}, \text{content} \]

(1) Vertex represents a vertex, type represents the type of the symbol, and its value is the English name of the graphic element, which is consistent with the element shown in Table 3; content is used to store the expression or variable declaration in the symbol.

(2) Arc represents the arc, verFrom and verTo respectively the start node symbol and the end node symbol of the arc; label is the label on the arc, and when Arc leaves the decision graph, it can only take "YSE" / "NO" value.

(3) AD represents the activity map, verSet is the vertex set in the activity diagram; arcSet is the arc collection in the activity diagram; entryVer indicates the entrance vertex of the activity diagram.

(4) ADModel represents the active graph model in the xUML model. ADSet is a collection of activity diagrams contained in the model; entryAD is used to record the entry activity diagram of the model.

3. xUML model to JAVA program conversion

3.1 Hierarchical Syntax Chart

The execution order of the activity graph is to traverse each vertex from the initial state to the termination state, but the sequential traversal cannot handle non-sequential structures such as branch structures, loop structures, and concurrent structures. To this end, a data structure called Hierarchical Syntax Chart (HSC) was introduced\(^7\)\(^8\) to over-convert xUML activity maps. First converting the activity diagram to the HSC model, and then converting the HSC model to the JAVA code. The HSC structure includes a composite node (com_stmt) and a common node (stmt).

```c
typedef struct com_stmt{
    string stmt_name;
    bool is_static;
    stmt_node* first_stmt;
    com_stmt* next_com_stmt;
}com_stmt_node, *HSC;
```

```c
typedef struct stmt{
    STMT_TYPE type;
    Vertex ver;
    com_stmt* first_com_stmt;
    stmt* next_stmt;
}stmt_node, *stmt_list;
```

The composite node (com_stmt) is mainly used to represent the execution branch of the ingress node and the non-sequential structure, and is represented by a circular symbol in the HSC model. Meanwhile, stmt_name is the name of the composite node, which can be the name of the function, or the name of the non-sequential structure execution branch, such as "YSE", "NO", and so on. The members first_com_stmt and next_com_stmt represent the first normal node and the next composite node, respectively. When the initial state is created to match the node, is_static is used to record whether the function is a static function method. The structure of HSC is shown in Fig 1.
3.2 Conversion algorithm

The conversion algorithm of xUML activity diagram to JAVA program is divided into two parts: xUML to HSC model conversion algorithm and HSC model to JAVA program conversion algorithm.

3.2.1 xUML model to HSC model conversion algorithm

The conversion algorithm of the xUML activity map to the HSC model is composed of functions ADMModel2HSC(), AD2ComStmt(), DFS_Trans(), Handle_Decision(), Handle_Parallel(), and Handle_Branch(), and each function is defined in the pseudo C language as follows. The ADMModel model is a parameter of the main function of the algorithm. It enumerates each activity graph in the xUML activity diagram model and calls the function AD2ComStmt to convert it into a compound statement. The returned statements are concatenated into a complete HSC.

The function AD2ComStmt() converts the active graph into a composite node based on the depth-first search algorithm (DFS) of the directed graph. To this end, two boolean arrays visited and isParent are designed to mark whether the vertex is accessed and whether the parent of the currently accessed vertex has been accessed. In addition, the stack S is used to maintain control flow order.

```
ADModel2HSC(ADMModel model) {
    com_stmt_node * hsc = NULL;
    Foreach ad in model {
        com_stmt_node cs_node = AD2ComStmt (ad);
    }
    return hsc;
}
```

```
AD2ComStmt (AD ad){
    bool visited[] = {false};
    bool isParent[] = {false};
    InitStack(S);
    ```

```
DFS_Trans (Vertex ver, bool visited[], bool isParent[], Stack S) {
    if(visited[ver]==true){
        return Handle_Branch(ver, isParent, S);
    }
    visited[ver]= isParent[ver] = true;
    newNode.first_com_stmt = NULL;
    newNode.ver = ver;
    switch(ver.type){
        case ActiveNode or ActionNode or FunctionCall:
           DFS_Trans(ver.nextVer,visited, isParent,S);
            ```
```

Fig 1. HSC structure
com_stmt node;
node.stmt_name =
ad.entryVer.content;
node.first_stmt = DFS_Trans
(ad.entryVer, visited, isParent, S);
return node;
}

Handle_Branch(Vertex ver, bool
isParent[], Stack S) {
nodeTop = GetTop(S);
if (isParent[ver] == true) {
nodeTop.type = TYPE_LOOP;
return NULL;
} else {
left = Split(nodeTop.first_com_stmt, ver);
nodeTop.next_stmt = left;
return left;
}
}

Handle_Decision(stmt newNode,
Vertex ver, Stack S) {
newNode.type = TYPE_IF;
push(S, newNode);
newComStmtYes.stmt_name = "YES";
newComStmtYes.next_com_stmt = NULL;
newComStmtYes.next_stmt =
DFS_Trans(GetBranch(ver, "YES"), visited, isParent, S);
newNode.first_com_stmt = newComStmtYes;
if (newNode.type == TYPE_LOOP) {
Pop(S);
newNode.next_stmt = DFS_Trans(GetBranch(ver, "NO"), visited, isParent, S);
} else {
newComStmtNo.stmt_name = "NO";
newComStmtNo.next_com_stmt = NULL;
newComStmtNo.next_stmt =
DFS_Trans(GetBranch(ver, "NO"), visited, isParent, S);
newComStmtYes.next_com_stmt = newComStmtNo;
Pop(S);
}
}

### 3.2.2 Conversion algorithm from HSC model to JAVA program

The algorithm for constructing a JAVA program from the HSC model is defined as follows. In this algorithm, the function GetVariables() is used to calculate the list of variables that appear in the active node and the action node; the function Trans_Exp() converts the xUML expression to its corresponding JAVA program expression; the function Trans_Stmt() declares the xUML Statements, assignment statements, etc. are converted into their corresponding JAVA program statements, and the function arraycode is used to arrange the generated code to be indented.

```java
HSC2JAVA(com_stmt_node* hsc) {
    com_stmt_node p = hsc;
    string code;
    while (p) { //Deal with each function
        code = code + p.stmt_name + "\n" + "+" + "\n" +
        CompStmt2JAVA(p) + "\n";
    }
    arrange(code);
    return code;
}

CompStmt2JAVA(com_stmt_node comStmt) {
    // Convert each statement to JAVA
    ...
p=comStmt.first_stmt;
varList = GetVariables(p);
string body ;
while(p){
    body = body + Stmt2JAVA(p) ;
    p = p.next_stmt;
}
return body;

case TYPE_IF:
    body="if("+Trans_Exp(node.ver.content)
+"\n"+"\n"+
    CompStmt2JAVA(node.first_com_stmt)
+"\n"
+"else"+"\n"+"\n"+
    CompStmt2JAVA(node.first_com_stmt.next
    _com_stmt)
+"\n"; 
    return body;
}

Then, the code should be indented by the algorithm. The arranging algorithm is described with
JAVA language as below:

```java
private String RearrangeCode(String code)
{
    int begin=-
1,end,i,j,k=0,rowCount=0,braceNum=0;
    String space,tmp;
    String[] message;//message is used to save
    the code
    for (i=0;i<code.length();i++)
        if (code.charAt(i) == 'n') // Find
           rowCount++;
    message=new String[rowCount];
    for (i=0;i<code.length()-1;i++)
        if (i == 0 || code.charAt(i) == 'n')// Find
            the location
                { 
                    for (j=i+1;code.charAt(j) != 'n';j++)
                        ;
                    end=j;
                    message[k]=code.substring(begin+1, 
end);
                    k++;
                    begin=end;
                    i=end-1;
                }

for (i=0;i<message.length;i++)
    { 
        tmp=message[i];// Keep a copy of the
        current information 
        if (tmp.equals("}"))
            braceNum--;
        space="";
        for (j=1;j<=4*braceNum;j++)
            space=space+" "; // Generate a fixed
            number of indents 
        message[i]=space+message[i]; // Increase space indentation 
        if (tmp.equals("{"))
            braceNum++;
    }
    code="";
    for (i=0;i<message.length;i++)
        code=code+message[i]+"\n";
    return code;
}
```

4. experiment
In this paper, we take the factorial problem of n as an example to verify the algorithm proposed in the
previous paper. This paper designs a code generation tool for xUML to JAVA program. After drawing
the xUML activity diagram of the program in the modeling module, the JAVA code of the program
can be generated. The xUML activity diagram modeling process is shown in Figure.2, and Figure 3
shows the generated code:
This paper proposes an extended xUML language based on the UML language and solves the problem of incomplete semantics of UML. Meanwhile, the method of this paper can generate JAVA programs. This paper designs the conversion algorithm from xUML activity diagram to HSC and HSC to JAVA program, and verifies it through experiments. In subsequent studies, the following questions will be analyzed in depth: (1) xUML static models, such as class diagram processing methods; (2) model integrity and consistency check problems; (3) xUML to process-oriented languages Conversion method.

5. conclusion

Fig. 2. Activity diagram

Fig. 3. JAVA code
Acknowledgments
This paper is supported by the Graduate Innovation Fund in Xi’an University of Posts and Telecommunications under Grant 103-602080002.

References:
[1] Albert M, Cabot J, Cristina Gómez, et al. Automatic generation of basic behavior schemas from UML class diagrams. Software and Systems Modeling. 9 47-67.
[2] Campos-Rebelo R, Pereira F, Moutinho F, et al. From IOPT Petri nets to C: An automatic code generator tool. IEEE International Conference on Industrial Informatics. 2011.
[3] Li Qingshan, Chen ping and Chu hua. Research and Application of Metadata Driven Model Supporting Flexible Mechanism. Journal of Xidian University. 29 319-323.
[4] Mcclure C. Relationship to the OMG Unified Modeling Language. 2001.
[5] Semantics of a Foundational Subset for Executable. Object Management Group, 2016.
[6] He Hongyue, Wang Zhixue, Wang Qinglong and Zhu Weixing. Modeling and Analysis of C4ISR Architecture Based on fUML. Systems Engineering and Electronics. 36 1875-1876.
[7] SHU X, WANG M and WANG X. Extending UML for Model Checking// SOFL+MSVL 2017. LNCS:10795. Berlin:Springer, 2018: 88-107.
[8] SHU X, Li C, Liu C. A Visual Modeling Language for MSVL// SOFL+MSVL 2016. LNCS:10189. Berlin:Springer, 2017: 220-237.