Consistent Code Generation from UML Models

Quan Long†‡[*], Zhiming Liu†[†], Xiaoshan Li§, and He Jifeng†[‡]
†International Institute for Software Technology, United Nations University, Macao, China
‡LMAM, Department of Informatics, School of Math., Peking University, Beijing, China
§Faculty of Science and Technology, University of Macau, Macao, China.
{longquan, lzm, hjf}@iist.unu.edu, xsl@umac.mo

Abstract

*Relational Calculus of Object Systems (r*COS *) is an OO-language which is equipped with an observation-oriented semantics and a refinement calculus based on the Hoare and He's Unifying Theories of Programming (UTP). In this paper, we give syntactic definitions for class diagrams and sequence diagrams in UML 2.0. Based on these definitions, we give an algorithm for checking the consistency of a class diagram and a sequence diagram. Furthermore, we develop an algorithm to generate r*COS *code from any given consistent class diagram and sequence diagram.*

Keywords. *Object Orientation, Semantics, UTP, UML, Code generation, Consistency*

1. Introduction

The Unified Modelling Language (UML) [4, 19] is now widely used in the development of software intensive systems. In a UML-based development process, such as the RUP [10, 11], several kinds of UML models are used to represent and analyze the artifacts created in a certain phase of the system development: class diagrams for static analysis (*static view*), state machines for dynamic behavioral specification and validation (*behavioral view*), sequence diagrams and collaboration diagrams for representing interactions between objects (*interaction*), OCL for specifications of functionalities and constraints of objects (*functional view*), etc . The UML multiview modelling has advantages. Each single view focuses on a different aspect so that the analysis and understanding of the various features of the modelled system can be done separately for that view [1]. The modeler is allowed to split a model of a system into several views to decompose it in chunks of manageable sizes. This is also important for tool development, including tools for code generation. However, a multiview model is confronted with the problem of consistency among the models of different views in a model of the whole system [16].

Researchers, e.g. [6, 1], have realized the conditions and solutions for consistency depending on the diagrams involved, the development process employed, and the current stage of the development. The difficulties in consistency checking lie in the fact that the syntax and semantics of UML are informal and imprecise compared with formal modelling notations. For example, many features including role names in class diagrams and object names in sequence diagrams are optional and may not appear in the diagrams. This causes no harm if UML is only used in its sketchy mode, but it is not satisfactory in the modes of blueprint and programming language [5], nor for consistent code generation. Also different models describe overlapping aspects of a system. This complicates the problem of consistency and the development of tools for code generation. Code generated from a model of the whole system should include information from sub-models for different views. But, according to our knowledge, current popular UML tools, such as Rational Rose [18, 3] and Fujaba [17], generate code only from the class diagrams and component diagrams. Other diagrams for behavioral and functional view, i.e. sequence diagram and state diagram, have been only used for analysis and testing of the system, but not integrated into the code. Therefore, those tools only generate code skeletons in which the methods generated only have signatures without their bodies.

Relational Calculus of Object Systems (rCOS)[7, 8] is a language for object oriented design. [1] It has a rich variety of features including subtypes, visibility, inheritance,

∗ Partially supported by National Natural Science Foundation of China (No. 60173003)
† On leave from the Department of Computer Science, the University of Leicester, UK.
‡ On leave from East China Normal University. Partially supported by the 973 project (2002CB312001)

1 In early publications, such as [7], the calculus for object-oriented design was named as OOL. rCOS is produced by LaTeXcommand {\large r}\textsc{COS}.

dynamic binding and polymorphism. The language is expressive enough for the specifications of object-oriented designs and programs. Further more, it is equipped with an observation-oriented semantics which is based on the Hoare and He's Unifying Theories of Programming (UTP) [9]. Based on the semantic model, we have investigated a calculus to support both structural and behavioral refinement of object-oriented designs. Based on the rCOS semantics, we have defined the semantics of requirement models and design models and the refinement relationships for UML models in our previous work [20, 21].

To compute the semantics defined in [20, 21], in this paper, we consider sequential software development and propose algorithms for consistency checking and rCOS -code generation. Code generation is carried out during the design stage when a design class diagram and a family of sequence diagrams are produced. By a design class diagram we mean a class diagram in which the classes are associated with their method signature declarations and associations have directions of visibility and navigation. For how to obtain such a design model from models requirement specification and analysis, we refer the readers to our early work [13, 12, 14, 15]. The design class diagram is used for the generation of a code skeleton. The sequence diagrams are used to generate the program text for method bodies in the skeleton. Code will be generated from a model only after the model passes the consistency checking. In this paper, we will consider the consistency and code generation from sequence diagrams and class diagrams. The code is a sequence of *class declarations* in rCOS .

To develop the algorithms, we first give syntactic definitions of class diagrams and sequence diagrams. Our definitions have taken the features of UML 2.0 which will cause the difficulties in the algorithm design into account. For example, the call back messages and the nested sequence diagrams. Our definition of sequence diagrams is compositional. So the users can choose to apply our algorithm to a part of the sequence diagram at a time. This adds flexibility to code generation that one does not have to generate the complete code. It also helps in terms of using system resource more efficiently.

The conditions for consistency between a class diagram and a sequence diagram were also formally defined in [15]. The correctness of the code generation algorithms can be semantically justified in the formal model for UML given in [12, 14, 15, 20].

The rest of this paper is organized as follows: We introduce the Relational Calculus of Object Systems (rCOS) in section 2 and give the syntactic definitions for class diagrams and sequence diagrams in Section 3. We present in Section 4 the algorithm for consistency checking. The algorithm for code generation is given in Section 5. In Section 6, we conclude our paper and the future work is discussed.

2. rCOS : A Language with Observation-Oriented Semantics

In this section we will introduce the main parts of the rCOS language with respect to the algorithm in this paper.

2.1. Syntax

In our model, an object system (or program) S is of the form *cdecls* • *P*, where *cdecls* is a *declaration* of a finite number of classes, and *P* is called the main method and is of the form $(\texttt{glb}, c)$ consisting of a finite set `glb` of *global variables* with their types and a command c. *P* can be understood as the *main method* if S is taken as a Java program.

2.1.1. Class declarations A declaration *cdecls* is of the form: $cdecls := cdecl \mid cdecls; cdecl$, where *cdecl* is a *class declaration* of the following form

$$\begin{array}{l} [\texttt{private}]\ \texttt{class}\ N\ \texttt{extends}\ M\ \{ \\ \quad (U_i\ u_i = a_i)_{i:1..m}; \\ \quad m_1(\underline{T}_{11}\ \underline{x}_1, \underline{T}_{12}\ \underline{y}_1, \underline{T}_{13}\ \underline{z}_1)\{c_1\}; \cdots; \\ \quad m_\ell(\underline{T}_{\ell 1}\ \underline{x}_\ell, \underline{T}_{\ell 2}\ \underline{y}_\ell, \underline{T}_{\ell 3}\ \underline{z}_\ell)\{c_\ell\}\} \end{array}$$

Note that

- A class can be declared as `private` or `public`. By default, it is assumed as `public`. We use a function *anno* to extract this information from a class declaration such that *anno*(*cdecl*) is *true* if *cdecl* declares a private class and *false* otherwise.
- *N* and *M* are distinct names of classes, and *M* is called the direct superclass of *N*.
- In our previous work [7, 8], attributes annotated with `private` are private attributes of the class, and similarly, the `protected` and `public` declarations for the protected and public attributes. We have these different kinds of attributes to show how visibility issues can be dealt with. Types and initial values of attributes are also given in the declaration. But in this paper we ignore such keywords for simplicity of the algorithm. We will add them in real toolkit development.
- The method declaration declares the methods, their value parameters ($\underline{T}_{i1}\ \underline{x}_i$), result parameters($\underline{T}_{i2}\ \underline{y}_i$), value-result parameters ($\underline{T}_{i3}\ z_i$) and bodies (c_i). We sometimes denote a method by $m(\underline{paras})\{c\}$, where $\underline{paras}$ is the list of parameters of m and c is the body command of m.
- The body of a method c_i is a command that will be defined later.

2.1.2. Commands Our language supports typical object-oriented programming constructs: [2]

$$c ::= \mathit{skip} \mid \mathit{chaos} \mid \textbf{var}\ T\ \text{x=e} \mid \textbf{end}\ x \mid c;c \mid c \lhd b \rhd c$$
$$\mid b * c \mid le.m(\underline{e},\underline{v},\underline{u}) \mid le := e \mid \mathit{C.new}(x)[\underline{e}]$$

where b is a Boolean expression, e is an expression, and le is an expression which may appear on the left hand side of an assignment and is of the form $le ::= x \mid le.a$ where x is a simple variable and a an attribute of an object. We use $le.m(\underline{e},\underline{v},\underline{u})$ to denote a call of method m of the object denoted by the left-expression le with actual value parameters $\underline{e}$ for input to the method, actual result parameters $\underline{v}$ for the return values, and value-result parameters $\underline{u}$ that can be changed during the execution of the method call and with their final values as return values too; and use the command $\mathit{C.new}(x)[\underline{e}]$ to create a new object of class C with the initial values of its attributes assigned to the values of the expressions in $\underline{e}$ and assign it to variable x. Thus, $\mathit{C.new}(x)[\underline{e}]$ uses x with type C to store the newly created object.

2.1.3. Expressions Expressions, which can appear on the right hand sides of assignments, are constructed according to the rules below.

$$e ::= x \mid \mathit{null} \mid \mathit{self} \mid e.a \mid e\ \texttt{is}\ C \mid (C)e \mid f(e)$$

where *null* represents the special object of the special class *NULL* that is a subclass of all classes and has *null* as its unique object, *self* will be used to denote the active object in the current scope (some people use `this`), $e.a$ is the a-attribute of e, $(C)e$ is the type casting, and e **is** C is the type test.

2.2. Semantics and Refinement Calculus

In [7, 8], we have developed an observation-oriented semantics for the above language using the basic model of the UTP. With the semantics we have developed a set of Object-Oriented refinement laws which covers not only the early development stages of requirement analysis and specification but also the later stages of design and implementation. Please see the details in [7] or [8].

In the rest of this paper we will investigate on how to formalize the class diagrams and sequence diagrams and how to transfer them into the rCOS -code.

3. Class Diagrams and Sequence Diagrams

3.1. Syntax of Class Diagrams

A class diagram CD consists of three parts:

2 In our previous work [7, 8], we also allow some commands for the purpose of specification and refinement. But we ignore them in the current version of code generation.

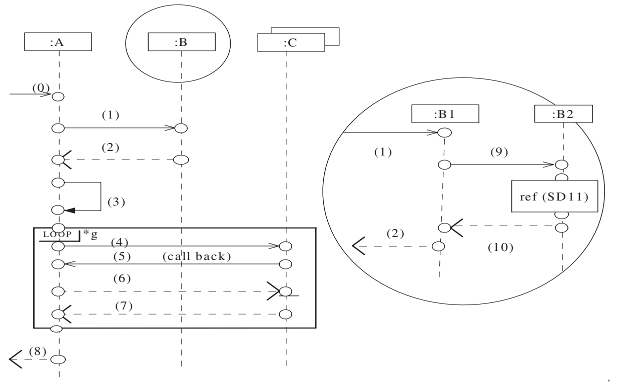

Figure 1. A sequence diagram

1. The first part provides the static information on classes and their inheritance relationships:
   - *CN*: the finite set of class names identified in the diagram. We use capital letters to represent arbitrary classes and types.
   - *super*: the partial function which maps a class to its direct superclass, i.e. $\mathit{super}(C) = D$ if D is the direct superclass of C.
2. The second part describes the structure of each class. Each $C \in CN$ is associated with two functions $\mathit{attr}(C)$ and $\mathit{meth}(C)$. $\mathit{attr}(C)$ maps C to a set $\{\langle a_1 : T_1\rangle, \cdots, \langle a_m : T_m\rangle\}$ of attributes, where T_i stands for the type of attribute a_i. $\mathit{meth}(C)$ maps C to a set of method signatures. Please note that at the first stage, $\mathit{meth}(C) = \{m_1()\{\}, \ldots, m_k()\{\}\}$. Our algorithm will generate the bodies and adds them into those $\{\}$s.
3. The third part identifies the associations among the classes: *AN*: the finite set of associations names captured in the diagram, and the function ass which maps the association name to its classes and the respective cardinalities: $\langle C_1 : card_1, C_2 : card_2\rangle$. For simplicity, we only deal with binary associations. General relations among classes can be modelled in the same way.

3.2. Syntax of Sequence Diagrams

We now give the syntactic definition of sequence diagrams. We will allow call back messages in a sequence diagram (e.g. message "(5)" in Figure 1). We also include some features of UML 2.0. For example, in a sequence diagram, we allow the combined fragment (except for the PAR one) [2], reference of other sequence diagrams and nested sequence diagram (i.e. each object of the sequence diagram can be another sub-sequence diagram.). The object "$\underline{:B}$" in Figure 1 represents a nested sequence diagram. The PAR combined fragment and asynchronous messages which will be handled in future work.

A sequence diagram *SD* consists of two main parts:

1. A sequence of objects: $\langle obj_1, obj_2, \cdots, obj_n \rangle$, which is denoted by *SD.Objects*. And each object obj_i has the following structure:
    - Each object *obj* is associated with a *type*, denoted by *type*(obj) which is either a class name C in *CN* or a sub-sequence diagram SD_1.
    - For an object *obj*, the property *multiob*(obj) equals *true* if *obj* is a multi-object (e.g. $\underline{:C}$ in Figure 1), otherwise *multiob*(obj) is *false*.
    - For an object *obj* there is a sequence of *time-points* $\langle p_1, p_2, \cdots, p_n \rangle$ which represents the time points during the lifetime of the object. These points represent the order of messages sending and receiving, the combination fragments and the references to other sequence diagrams. In Figure 1, small circles represent time points.

      We have a function *event* for each time-point to describe what happens at that time. For each point p ,

      $$event(p) \in \{send, ack, receive, receiveack, option, loop, endfrag, ref, endref\}$$

2. A set *MSG* of messages: each message *msg* is of the form $(source, m, target)$ where
    - *source*, denoted by *src*(msg), is a pair (obj, p) of an object and a time point. $src(msg) = (obj, p)$ means that object *obj* is the source of the message that occurs at time point p. We use *src*(msg)*.obj* and *src*(msg)*.p* to denote them respectively.
    - *target*, denoted by *tgt*(msg), is also a pair (obj, p) of an object and a time point.
    - m, denoted by *method*(msg), can be a method call of the form $(ass, method())$ (sometimes it is simply written as $ass.m()$), that represents that method *method*() of the target object is called by the source object via the association *ass*.[3] Also, m can be a command, such as an assignment, or any composite command, other kinds, but we require in this case the source object and the target object must be the same. This represents the execution of an internal action of the object. Finally, a message can be a return signal and in this case m is $\emptyset$.

    Potentially, the target or source point may be empty. If the target point is empty, it means the message is an outgoing message of the whole sequence diagram (e.g. message "(8)" in Figure 1). And if the source is, it means it is an incoming message (e.g. message "(0)" in Figure 1).

3 For simplicity of the paper, in the syntax, we ignore the parameters of methods. They can be easily added in the tool development.

Let us explain the definition as follows by using Figure 1.

- *event*(p)=*send* means a message (or equivalently, a method call) is sent out from this position of the current object. Similarly, *receive* means a message reaches at this position of the object. The message between a *send* point and a *receive* point will be drawn as a solid line arrow in the graph. For example, the message "(1)" and "(4)" in Figure 1.
- *ack* and *receiveack* points are used to denote the returned messages which are dotted line with open arrowhead back to the sending lifeline. For example, the message "(2)" and "(7)" in Figure 1. If a message *msg* is a returned one, then *method*(msg)=$\emptyset$.
- *option* and *loop* are used to represent the combined fragments which are the new features of UML 2.0.

  The option combination fragment is used to represent a sequence that, given a certain condition (guard), will be executed; otherwise, the sequence will not be executed. An option combination fragment is used to model a simple "if-then" statement. The loop combination fragment is used to represent a repetitive sequence. Given a guard, the body of the fragment will continue executing repetitively until the guard condition becomes false.

  Here *option* and *loop* represent the beginning of the option combination fragment and loop combination fragment respectively. If an *event* of a point is *option* or *loop*, it will be equipped with another function, $guard()$, which maps the point to its *guard* expression. A point with *endfrag* event represents the end of a combination fragment.
- *event*(p) is *ref* means that from point p the current sequence diagram begins to call another sequence diagram and *endref* represents the end of the call. A *ref* point will be equipped by a *name* representing the sequence diagram it calls.

4. Consistency Checking of a Class Diagram and a Sequence Diagram

Before generating the code from a class diagram and a sequence diagram, we need to ensure that the diagrams are consistent. We present an algorithm in this section for checking the consistency between a sequence diagram and an existing class diagram. The output of this algorithm is a file of a report on all inconsistencies found by the algorithm.

Our algorithm will only work correctly for *well-formed* sequence diagrams and class diagrams. The main condition of the well-formedness of a class diagram is that the inheritance relation does not introduce cycles between classes. Also we do not deal with multiple inheritance in a class diagram. The other issues are mainly naming problems. However, the well-formedness of a sequence diagram is a bit more complicated and it concerns the following conditions:

- For each message *msg* in the sequence diagram, the *event* of the source point of *msg* must be a *send* or *ack* and the *event* of target point of *msg* must be a *receive* or *receiveack*, respectively.
- If a point p_1 represents the beginning of combined fragments, i.e. *loop* or *option*, there must be one and exactly one corresponding *endfrag* point p_2 on the same object such that p_2 is later than p_1.
- For a point p_1 with *event*$(p_1) =$ *ref*, there must be an *endref* point p_2 on the same object such that p_2 follows p_1. The well-formedness of the sub-sequence diagram is checked recursively.
- If obj is a nested object, then for every *matched* pair of sending and returning messages $\langle(source, m, obj), (obj, \emptyset, target)\rangle$ in *obj*, there is a corresponding matched pair $\langle(\emptyset, m, target_1), (source_1, \emptyset, \emptyset)\rangle$ of source-less (incoming) message and target-less (outgoing) message in the subsequence diagram $type(obj)$. The order of these messages are preserved in the sub-sequence diagram. Finally the sub-sequence has to be well-formed.

The well-formedness of a sequence diagram has also to ensure the sequence diagram indeed represents a scenario of method calls. This means that (a). order of the message sending and receiving must be consistent, and for all messages from the same object, the earlier it is sent, the earlier it is received by the target object; (b). if a message *msg* invokes message *msg*$_1$, then *msg*$_1$ must return before *msg* does. To check this, we can use a token to traverse in the sequence diagram via the message arrows. The sequence diagram is well-formed with regard to this aspect, if the token starts at the first point of the first object and comes to the last point of the first object in the end.

The algorithms for checking the well-formed conditions are relative easy and we leave them out of this paper.

For a well-formed class diagram and a well-formed sequence diagram, our algorithm for consistency checking takes care of the following conditions. A violation of any of them will be reported as an inconsistency.

- **Association.** For each *msg* $\in SD$.*MSG* there must exist a corresponding association in the class diagram. Notice, this is static as it does not ensure that the object which is sending the message in a particular state during the execution is currently associated with the target object of the message.
- **Class Name.** For the above mentioned association, the two related classes in the class diagram must have the same names with the objects related to *msg* in the sequence diagram respectively.
- **Method.** The names and signatures of all methods in the sequence diagram must be the same with the ones in the class diagram. Further more, if an $m()$ is the method of a message sent from $\underline{:C}$ to $\underline{:D}$ in the sequence diagram, then $m()$ must be a method of class D in the class diagram.
- **Attribute.** The variables used in the guard of a combined fragment should be directly accessible by the invoking object.
- **Multiplicity.** If the association of class diagram is one to many, the corresponding object in the sequence diagram must be a multi-object. Notice here, other general class invariants should be ensured by the design of the sequence diagram and not by the consistency checking.

Our algorithm employs the Breadth First Search (BFS) strategy for nested SD. The main idea in the design of the algorithm is as follows.

- A queue *Queue* is used here, with three standard operations: *InQueue*(), *OutQueue*(), *QueueIsEmpty*().
- We have a method *FindAss*(ass) to search an association in the class diagram. If succeeded, it returns *ture*, otherwise, *false*.
- The inputs to the algorithm are a well-formed sequence diagram SD and a well-formed class diagram CD.
- The output of the algorithm is a report file *Inconsistency* containing all inconsistencies found. For this purpose, we have a special key word "PRINTFILE" which will print the inconsistent information to the file *Inconsistency*.

```
ALGORITHM BEGIN
VAR SD1:=SD;
WHILE (true) DO
  FOREACH i: $obj_i$ ∈SD1.Objects
    IF (type($obj_i$) is another sub-sequence diagram)
      InQueue($obj_i$); // BFS, nested SD enters Queue
    ENDIF
  ENDFOR      // begin checking SD1
  FOREACH msg: (msg∈SD1.MSG)∧event((scr(msg))=receive)
    VAR ass:=method(msg).ass;
 VAR exists:=FindAss(ass);
    IF (¬exists)
       PRINTFILE(msg+":Association Error");
     ELSE       IF ¬((type(src(msg).obj)=ass.$C_1$)
             ∧(type((tgt(msg).obj))=ass.$C_2$))
       PRINTFILE (msg+":Class Mismatch");
      ELSE
       IF method(msg)∉ meth(type(tgt(msg).obj));
         PRINTFILE (method(msg)+
                     ":Method does NOT exist");
       ENDIF
```

```
        IF(ass.$card_1$=multi∧Multiob((src(msg).obj)=false)
          PRINTFILE ((src(msg).obj+;
                      ":Multiplicity Error)"
        ENDIF
        IF(ass.$card_2$=multi∧Multiob((tgt(msg).obj)=false)
          PRINTFILE ((tgt(msg).obj+;
                      ":Multiplicity Error")
        ENDIF
      ENDIF
     ENDIF
    ENDIF
  ENDFOR
  FOREACH point:(point∈SD1)∧(event(point)∈ {loop,
                                            option})
    IF (The variables used in guard(point) is
         out of the defined scope)
      PRINTFILE (guard(point) ": Invalid Reference");
    ENDIF
  ENDFOR              // finish checking SD1
  IF (¬QueueIsEmpty())
    SD1:=OutQueue();
  ELSE RETURN;
  ENDIF
ENDDO
ALGORITHM END
```

5. Algorithm for Code Generation

We have already checked the static consistency between a sequence diagram and a class diagram. Now in this section we investigate an algorithm for generating the *class declaration* segment in rCOS code from them.

The *class declaration* to be generated will declare a class for each class and association in CD. If a method of object C_1 is called by object C_2, it will declare an attribute for the reference of C_1 in class C_2. Each multi-object in SD will have a special class to handle it. We use the *Vector* type to represent a sequence of object instances. So when we map it to the real Java code in future we can use the system-offered *add*(), *delete*() and *insert*() methods.

Note that the algorithm generates the method bodies from the sequence diagram. Figure 1 illustrates the method bodies as follows: message "(1)" represents a method call and message "(2)" is its corresponding returning message. The sending message between the end points of them, i.e. "B2.(9)" is the body of the method "*method*((1))". The result of this method body is in class `B1` in the example at next page. To achieve such results, we design the strategy of our algorithm as follows. Our algorithm traverses the sequence diagram during the generation of method bodies. (We have *call back*s, so the algorithm has to do so.) For a *send* point, if it calls m, the algorithm writes the signature of m to the body of the method that is currently being generated, leaving the method body unfinished, begins to write the body of method m. We use a stack to store the methods for which the generation of their bodies have started but not yet completed. The top element of the stack is the method currently being generated. The algorithm traverses through the sequence diagram and the top element is popped out when its return message, i.e. the corresponding *ack* message, is visited. At a point which is not a *send* or an *ack*, the execution of the algorithm goes down through the lifeline and generates the corresponding code for "if", "while" statements or the "commands".

In this algorithm we also use the Breadth First Search (BFS) strategy for nested sequence diagrams. We give an outline of the algorithm as follows.

- A queue *Queue* is used, with the standard operations, *InQueue*(), *OutQueue*() and *QueueIsEmpty*().
- A stack *Stack* is used to store the information about the methods for which the code generation has started but not yet completed. Again, we use the standard stack operations: *Push*(), *Top*(), *Pop*() and *StackIsEmpty*().
- We have a method *next*() which maps each point to its next point at the same object lifeline. If the point itself is the last one, then the mapping is undefined. Please notice that, in our algorithm, we can make sure that the method *next*() will not be invoked on the last point during the traversing.
- For each point p, we use $message(p)$ to denote the message sent or received at p.
- The output is a text file of class declarations. We have two key words "WRITEFILE" and "WRITECLASS" to write the file. The key word "WRITECLASS (*classname*)" means locating the writing pointer to the *classname*'s class declaration segment (It can be implemented by the *Hash table* in Java.). And "WRITEFILE(...)" means to write the content between the parentheses to the file in the place where the writing pointer locates.
- Because we use BFS, we need to generate a class declaration for an object *obj* with *type*(*obj*) as a sequence diagram. However, when we come back to generate the class declarations for the sub-sequence diagram of obj, we need to delete the original class declaration for *obj*. We use the keyword "DELEFILE" for this purpose.
- The input to the algorithm is a pair of consistent sequence diagram SD and class diagram CD.

  The output is a file containing a sequence of class declarations (as shown in the later example that is generated from the sequence diagram in Figure 1).

The details for the algorithm is in the *appendix A* of this paper.

Here we give an example for the sequence diagram depicted in Figure 1. For simplicity, we suppose the corresponding class diagram has no inconsistent issue and we ignore the attributes, associations in the class diagram. We use `meth_i()` to represent the signature of *method* (*message*(i)). The rCOS code generated from Figure 1 is as follows:

```
class A {              |  class C-handler{
 B refB1;              |   C element;
 C refC;               |   Vector<C> vector;
 meth_0(){             |  }
  refB1.meth_1();      |
  command_3;           |  class C{
  (g)* {               |   A refA;
   refC.meth_4();      |   meth_4(){
  }                    |    refA.meth_5();
 }                     |   }
 meth_5(){}            |  }
}                      |
                       |
class B1{              |  class B2{
 B2 refB2;             |   meth_9(){
 meth_1(){             |    SD_11();
  refB2.meth_9();      |   }
 }                     |  }
}                      |
```

The complexity of this algorithm is $O(n)$ where n is the number of messages. For readability of the algorithm we ignore some implementation details. For example, if an object obj_1 calls another object obj_2 more than once, the algorithm will generate several same references of obj_2 in obj_1 which is not permitted in rCOS . If an object call itself, the similar problem will appear. We will deal with such non-essential issues in the toolkit development.

In our earlier work [12, 15], the design model is composed of a class diagram and a set of sequence diagrams. We can extend the algorithm of this paper for checking the consistency between sequence diagrams, and then apply the code generation algorithm to the consistent sequence diagrams one by one to generate the class declarations for the design model.

6. Conclusions and Future Work

In this paper, we have syntactically defined class diagrams and sequence diagrams. The definition for sequence diagrams covers most features of UML 2.0. Based on these definitions we developed an algorithm for checking the consistency between a class diagram and a sequence diagram. Furthermore, we have proposed an algorithm to generate a sequence of rCOS class declarations from a consistent pair of a class diagram and a sequence diagram. The class declarations include the program texts for the bodies of the methods that appear in the sequence diagram.

With the algorithms in this paper, we can compute the semantics of requirement models and design models defined in our previous work [20, 21]. And further more, in the UML-based software development, we can prove (by computing) the refinement relationship for UML models using the refinement laws in [7, 8]. We will develop a tool to support the above mentioned usages.

The semantic correctness of the code generation can be justified by our earlier and ongoing work on formal semantics of UML models [13, 12, 14, 15]. In fact, we have defined the semantics of the generated code [20, 21] and can prove it is correct according to the semantics of UML models given in [14, 15].

The rCOS code is very similar to Java. We will develop a toolkit based on the algorithms to generate real Java code. For simplicity, we do not consider the public, private keywords for attributes and methods in this paper. We will consider them in later toolkit development.

In future work, we will consider the asynchronous messages in sequence diagrams. We will give the semantics to sequence diagrams with respect to the dynamic consistency checking between statechart diagrams and sequence diagrams. Further more, we will give the algorithm to generate code which includes the information of the statechart diagrams of the system.

Acknowledgement The first author would like to show his appreciation to Zongyan Qiu and Yifeng Chen for fruitful discussions. Xiaoqing Peng and Yuxin Cheng have carefully read the draft of this paper and gave some useful comments.

References

[1] E. Astesiano and G. Reggio. An attempt at analysing the consistency problems in the UML from a classical algebraic viewpoint. In *Proc. WADT 2002, LNCS 2755*. Springer Verlag, 2003.

[2] Donald Bell. UML's sequence diagram. Technical report, IBM, http://www-106.ibm.com/developerworks/rational/library/3101.html, 2004.

[3] W. Boggs and M. Boggs. *Mastering UML with Rational Rose 2002*. SYBEX, 2002.

[4] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modelling Language User Guide*. Addison-Wesley, 1999.

[5] M. Fowler. What is the point of UML. In *LNCS 1618*. Springer, 1998.

[6] J.M. Kuester G. Engels and L. Groenewegen. Consistent interaction of software components. *Pro. of IDPT2002*, 2002.

[7] J. He, Z. Liu, X. Li, and S. Qin. A relational model for object-oriented designs. In *Pro. APLAS'2004, LNCS 3302*, Taiwan, 2004. Springer.

[8] J. He, Z. Liu, X. Li, and S. Qin. A relational model for object-oriented designs. Technical Report 308, UNU/IIST, P.O. Box 3058, Macao SAR China, 2004.

[9] C.A.R. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.

[10] I. Jacobson, . Rumbaugh, and G. Booch. *The Unified Modelling Language Reference Manual*. Addison-Wesley, 1999.

[11] P. Krichten. *The Retional Unified Process - An Introduction*. Addison-Wesley, 2000.

[12] J. Liu, Z. Liu, J. He, and X. Li. Linking UML models of design and requirement. In *Pro. ASWEC'2004*, Melbourne, Australia, 2004. IEEE Computer Sciety.

[13] Z. Liu, J. He, X. Li, and Y. Chen. A relational model for formal requirements analysis in UML. In J.S. Dong and J. Woodcock, editors, *Formal Methods and Software Engineering, ICFEM03, LNCS 2885*, pages 641–664. Springer, 2003.

[14] Z. Liu, J. He, X. Li, and J. Liu. Unifying views of UML. Technical Report 288, UNU/IIST, P.O. Box 3058, Macao SAR China, 2004.

[15] Z. Liu, X. Li, J. Liu, and J. He. Integrating and refining UML models. Technical Report 295, UNU/IIST, P.O. Box 3058, Macao SAR China, 2004.

[16] S.J. Mellor and M.J. Balcer. *Executable UML: a foundation for model-driven architecture*. Addison-Wesley, 2002.

[17] Susannah Moat. Fujaba code generation for static UML models. Technical report, University of Paderborn, http://wwwcs.upb.de/cs/fujaba/documents/developer /guidelines/codegenerierung.pdf, Dec. 2002.

[18] T. Quatrani. *Visual Modeling with Rational Rose 2000 and UML*. Addison-Wesley, 2000.

[19] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Software Development Process*. Addison-Wesley, 1999.

[20] J. Yang, Q. Long, Z. Liu, and X. Li. A predicative semantic model for integrating UML models. In *Pro. ICTAC'2004, LNCS 3407*, Guiyang, China, 2004. Springer.

[21] J. Yang, Q. Long, Zhiming Liu, and X. Li. A predicative semantic model for integrating UML models. Technical Report 309, UNU/IIST, P.O. Box 3058, Macao SAR China, 2004.

Appendix A

The algorithm for code generation:

```
ALGORITHM BEGIN
VAR SD1:=SD;
FOREACH A: A∈CD.AN // write association classes
  WRITEFILE("class "+A+"{role1 "ass(A).$C_1$ +";"
                      +" role2 "ass(A).$C_2$+ ";}");
ENDFOR
WHILE (true) DO
  FOREACH $obj_i$: $obj_i$ ∈SD1.Objects
    IF (type($obj_i$) is a sub-sequence diagram)
      InQueue($obj_i$); // BFS, nested SD enters Queue
    ENDIF
  ENDFOR
    // begin translate(SD1)
  FOREACH i: $obj_i$ ∈ SD1.Objects // write some classes before traverse
    IF (Multiob($obj_i$)=true)
      WRITEFILE ("class "+type($obj_i$)+"-handler {"
                 +type($obj_i$)+ "element;"
                 +"Vector ⟨"type($obj_i$)+"⟩"+" vector;}");
    ENDIF          // a special handler class for multi-objects
    IF (super(type($obj_i$))$\neq \emptyset$)
      WRITEFILE ("class "+type($obj_i$)+" extends
               "+super(type($obj_i$))+"{}");
    ELSE WRITEFILE ("class "+type($obj_i$)+" {}");
    ENDIF          // declare the class for $obj_i$
    WRITECLASS (Currentobj);
    FOREACH j: $\langle a_j : T_j \rangle \in attr(type(obj_i))$
      WRITEFILE ($T_j$+" "$a_j$+";");
    ENDFOR          // declare attributes in $CD$
  ENDFOR
    // traverse begins
  VAR Currentobj:=SD1.$obj_1$;// the first object of SD1
  VAR Currentpoint:=Currentobj.$p_1$;
                  // the first point of the first object
  WRITECLASS (Currentobj);
  WRITEFILE(method(message(Currentpoint))+"{");
  Push(Currentpoint);          // first point enters stack
  Currentpoint:=next(Currentpoint);
  WHILE(¬StackIsEmpty()) DO
    IF (Currentpoint.obj=$obj_1$)∧(next(Currentpoint)=∅))
      WRITECLASS ($obj_1$); WRITEFILE ("}");
      BREAK;
        // meeting the last returned message means the end of translating SD1
    ENDIF
    INCASE event(Currentpoint)=
      CASE send  BEGIN
          Currentobj:=Top().obj;
          Push(Currentpoint);
          WRITECLASS (Currentobj);
          VAR obj1:=tgt(message(Currentpoint)).obj;
          WRITEFILE (type(obj1)+" ref"+type(obj1) +";");
         // add reference of the called object as attribute in the calling object
          WRITEFILE(obj1+"."+method(message(Currentpoint))
                +";");
         // add the called method to the body of the current method
          Currentpoint:=tgt(message(Currentpoint)).point
          Currentobj:=obj1;
          WRITECLASS (Currentobj); //relocate the writing pointer
          WRITEFILE (method(message(Currentpoint))
                        +"{");
              // continue traverse SD1 and start writing the next method body
        END
      CASE ack   BEGIN
          Currentpoint:=tgt(message(Currentpoint));
                        // go back to the calling method
          WRITEFILE("}");    Pop();
                // finish writing the current method
          Currentpoint:=next(Currentpoint);
        END
      CASE ref  BEGIN
          WRITEFILE(ref.name+"();");
          Currentpoint:=next(Currentpoint);
        END
      CASE loop  BEGIN
          WRITEFILE ( guard(Currentpoint)+"*{");
          Currentpoint:=next(Currentpoint);
        END
      CASE option  BEGIN
        WRITEFILE("skip◁"+ guard(Currentpoint)+"▷{");
          Currentpoint:=next(Currentpoint);
        END
      CASE endfrag  BEGIN
          WRITEFILE("}");
          Currentpoint:=next(Currentpoint);
        END
      DEFAULT
          Currentpoint:=next(Currentpoint);
    ENDCASE // the end of handling this point
  ENDDO      // the end of the translating SD1
  FOREACH $obj_i$: $obj_i$ ∈SD1.Objects
    IF ($obj_i$ is a sub-sequence diagram) DELEFILE($obj_i$);
    ENDIF
        // delete the nested classes which will be written later in the algorithm.
  ENDFOR
  IF (¬QueueIsEmpty())
    SD1:=OutQueue();
  ELSE RETURN;
  ENDIF
ENDDO
ALGORITHM END
```