The United Nations University

UNU/IIST

International Institute for Software Technology

A Formal Semantics of UML Sequence Diagrams

Xiaoshan Li, Zhiming Liu and He Jifeng

February 2004

UNU-IIST Report No. 292

UNU-IIST and UNU-IIST Reports

UNU-IIST (United Nations University International Institute for Software Technology) is a Research and Training Centre of the United Nations University (UNU). It is based in Macau, and was founded in 1991. It started operations in July 1992. UNU-IIST is jointly funded by the Governor of Macau and the governments of the People's Republic of China and Portugal through a contribution to the UNU Endownment Fund. As well as providing two-thirds of the endownment fund, the Macau authorities also supply UNU-IIST with its office premises and furniture and subsidise fellow accommodation.

The mission of UNU-IIST is to assist developing countries in the application and development of software technology.

UNU-IIST contributes through its programmatic activities:

1. Advanced development projects, in which software techniques supported by tools are applied,
2. Research projects, in which new techniques for software development are investigated,
3. Curriculum development projects, in which courses of software technology for universities in developing countries are developed,
4. University development projects, which complement the curriculum development projects by aiming to strengthen all aspects of computer science teaching in universities in developing countries,
5. Schools and Courses, which typically teach advanced software development techniques,
6. Events, in which conferences and workshops are organised or supported by UNU-IIST, and
7. Dissemination, in which UNU-IIST regularly distributes to developing countries information on international progress of software technology.

Fellows, who are young scientists and engineers from developing countries, are invited to actively participate in all these projects. By doing the projects they are trained.

At present, the technical focus of UNU-IIST is on formal methods for software development. UNU-IIST is an internationally recognised center in the area of formal methods. However, no software technique is universally applicable. We are prepared to choose complementary techniques for our projects, if necessary.

UNU-IIST produces a report series. Reports are either Research R, Technical T, Compendia C or Administrative A. They are records of UNU-IIST activities and research and development achievements. Many of the reports are also published in conference proceedings and journals.

Please write to UNU-IIST at P.O. Box 3058, Macau or visit UNU-IIST's home page: http://www.iist.unu.edu, if you would like to know more about UNU-IIST and its report series.

Chris George, Acting Director

The United Nations University

UNU/IIST

International Institute for Software Technology

P.O. Box 3058
Macau

A Formal Semantics of UML Sequence Diagrams

Xiaoshan Li, Zhiming Liu and He Jifeng

Abstract

This paper presents a formal semantics of UML sequence diagram. In abstract syntax form, a well-formed sequence diagram corresponds to an ordered hierarchical tree structure. The static semantics of a sequence diagram is to check whether it is consistent with the class diagram declaration as well as with its well-formed tree structure. Meanwhile, the dynamic semantics is defined in terms of the state transitions that are carried out by the method invocations in the diagram. When a message is executed, it must be consistent with system state, i.e., object diagram and the state diagrams of its related objects. The semantics clearly captures the consistency between sequence diagram with class diagram and state diagram. Therefore, it is useful to develop the model consistent checking functions in UML CASE tools. And it also can be used to reason about the correctness of a design model with respect to a requirement model.

This paper is to occur in the proceedings of ASWEC2004, 13-16 April, 2004, Melbourne, Australia.

Xiaoshan Li is an Associate Professor at the University of Macau. His research areas are interval temporal logic, formal specification and simulation of computer systems, formal methods in system design and implementation. E-mail: xsl@umac.mo.

Zhiming Liu is a Research Fellow at UNU/IIST, on leave from Department of Computer Science at the University of Leicester, Leicester, England where he is lecturer in computer science. His research interests include theory of computing systems, emphasising sound methods for specification, verification and refinement of fault-tolerant, real-time and concurrent systems, and formal techniques for OO development. His teaching interests are Communication and Concurrency, Concurrent and Distributed Programming, Internet Security, Software Engineering, Formal specification and Design of Computer Systems. E-mail: Z.Liu@iist.unu.edu.

He Jifeng is a Senior Research Fellow of UNU/IIST. He is also a professor of computer science at East China Normal University and Shanghai Jiao Tong University. His research interests include the Mathematical theory of programming and refinement methods, design techniques for the mixed software and hardware systems. E-mail: hjf@iist.unu.edu.

Contents

1 Introduction

The Unified Modeling Language (UML) [BRJ99, RJB99] is a general-purpose visual modelling language that is used to specify, visualize, construct, and document the artifacts of a software system. It has become the de-facto standard for object-oriented modelling. This paper presents a formal semantics of sequence diagram by using a slightly modified version of the classic model of *transition systems* (or *action systems*) [MP81]. The motivation is to provide a natural and intuitive formal semantics of UML sequence diagram based on the users' general understanding of object-oriented programming language.

In fact, there are nine kinds of diagrams in UML for modelling system from different aspects. The system functional requirements are captured in use case diagrams. In order to realize use cases, system analysts and designers should provide system static structural models: class diagrams and system dynamic models: sequence diagrams (collaboration diagrams), state diagrams, and activity diagrams. An object diagram is an instance of a class diagram. Component and deployment diagrams are used for modelling system management and architecture aspects. However, the following two problems are most important when people are concerned with system analysis and design process with UML:

- How can we ensure that the models for a system analysis and its design are consistent?
- How can we check that a design model correctly realizes a system requirement model?

The first question asks us to check the consistency between different models. For example, if a sequence diagram uses an object, the class (or type) of the object should be defined in class diagram. As for the second, we must guarantee that each use case can be realized by its corresponding sequence diagrams in the context of the design class diagram.

In order to answer the two questions above clearly, we need a formal semantics for UML different diagrams. In this paper, we only focus on the semantics of sequence diagrams as well as its relevant problems with other diagrams. Based on the formal semantics of sequence diagram, we can answer questions like:

- Is a sequence diagram well defined?
- Is a sequence diagram consistent with a class diagram?
- Is a sequence diagram consistent with a state diagram?
- Does a sequence diagram correctly realize a use case of which formalization was given in [LLH02]?

Such a semantics is also required for the development of tools for identifying inconsistencies.

In this paper, we define a static semantics for UML interaction diagrams to support checking the well-formedness of an interaction diagram. This well-definedness is defined and checked in the context of other diagrams, i.e. its consistency with a class diagram and a state diagram. We also define a dynamic semantics to capture the behavior of an interaction diagram as a finite sequence of message calls.

Since we only consider the synchronous method call, the event of sending and receiving signals is ignored in this paper. We directly interpret the interaction between objects into object method invocation in object

oriented programming language sense. Concurrency with asynchronous communication can be considered in an extension of this semantic model for component-based development [HLL03]. Therefore, the object, method call and other concepts of object-oriented programming language can be used as basic concepts for defining the semantics of sequence diagram.

The dynamic semantics of a sequence diagram in this paper is interpreted as a trace-based terminated process (thread) of CSP. The advantage of this semantics is that it can help system analysts and designers to draw interaction diagram based on their understanding of OO methodology and OO programming. Therefore, it will be natural to translate the UML models to Java code generation.

The remainder of this paper is organized as follows. Section 2 introduces the notions of class diagrams and object diagrams and their representation in our framework. Section 3 overviews the UML sequence diagram as well as the relationship with other UML models, i.e., class diagrams and state diagrams. The syntax of sequence diagram is defined formally. In Section 4, we introduce the computational model that we will use to define sequence diagrams. The formal semantics of sequence diagram is given in Section 5 from both static and dynamic aspects. Finally, the conclusions are drawn in Section 6 with some discussions on the semantics as well as related and future work.

2 Class Diagrams and Object Diagrams

As we will define the semantics of a sequence diagram in the context of a class diagram, we introduce the notation of class diagrams first. A class diagram Δ of an application identifies the environment in which the sequence diagrams operate. This environment consists of four parts:

1. The first part provides the *static* information on classes and their inheritance relationships:
   - CN: the finite set of classes 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. **super**$(C) = D$ if D is the direct superclass of C.
2. The second part describes the structure of each class and for $C \in CN$, it includes

   $attr(C)$: the set of $\{< a_1 : T_1 >, \ldots, < a_m : T_m >\}$ attributes of C, where T_i stands for the type of attribute a_i of class C, and will be referred by $type(\mathrm{C}.a_i)$.

   As in [AM02], the type of an attribute is assumed to be a built-in *simple data type*, such as the natural numbers $\mathbf{Nat}$. Each class C defines a type, also denoted by C. We allow the construction of a type from the direct product of two types, and the power set $\mathbb{P}(T)$ of a type T.
3. The third part identifies the relationships among the classes:

   AN: the finite set of associations names captured in the diagram and the set Ass of associations that is a subset of $CN \times AN \times CN$.

   For simplicity, we only deal with binary associations. General relations among classes can be modelled in the same way.
4. The fourth part defines the set of methods for each class in the diagram:

   $method(C)$ is the set of all methods of class C in the diagram.

The multiplicities of the roles of an association will be specified in the invariant of the system [LLH02] which are not essential to this paper.

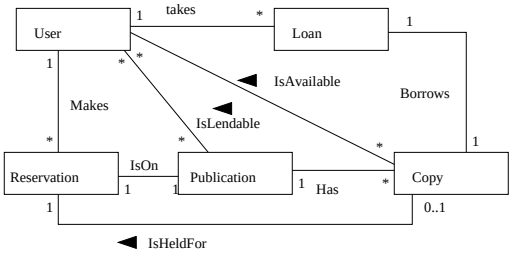

Figure 1: Conceptual class diagram of a library system

Example The formalization of a conceptual class diagram of a library system shown in Figure 1 is given as follows, where we omitted the methods of classes, and every class is a subclass of class $Object$ and we omit $attr(C)$ when C has no attributes:

$$
\begin{aligned}
CN = \ & \{User, Loan, Copy, \\
& Publication, Reservation\} \\
AN = \ & \{Takes, Borrows, IsAvailable, \\
& IsLendable, Has, IsOn, \\
& IsHeldFor, Makes\} \\
\mathbf{super}(C) = \ & Object, \text{ for all } C \in CN \\
attr(User) = \ & \{< id : String, name : String >\} \\
attr(Loan) = \ & \{< date : Date >\} \\
Ass = \ & \{< User, Takes, Loan >, \\
& < Loan, Borrows, Copy >, \\
& < Copy, IsAvailable, User >, \\
& < Publication, IsLendable, User >, \\
& < Publication, Has, Copy >, \\
& < Reservation, IsOn, Pulication >, \\
& < Copy, IsHeldFor, Reservation >, \\
& < User, Makes, Reservation >\}
\end{aligned}
$$

A refinement of the model allows us to add more details, such as attributes and associations [HLL02].

The well-formedness of a class diagram is specified in terms of OCL, that can be easily defined in a relational semantic model [LLH02].

The dynamic semantics of a sequence diagram is to be defined in terms of state changes of the system. A *state* of the system with a given class diagram is in fact a UML object diagram of the class diagram. An object diagram is a snapshot of the class diagram which consists the information about what the current objects are, what the current links by associations among these objects are, and what the state of the objects are, i.e. the values of their attributes.

3 Sequence Diagrams

In UML use case diagram, each use-case is a system function. Generally, a use case is defined as one basic course and several alternate courses in requirement analysis phase. A use-case course describes a sequence of interactions between actors with the system, which is a abstract template of a family of scenarios. According to each course description, an sequence diagram is presented in system design phase to realize the corresponding use case course.

Sequence diagrams are used to present the dynamic behavior of system design while class diagrams are system static structure. As one of two kinds of UML interaction diagrams, a sequence diagram shows interactions between objects arranged in a time sequence.

A sequence diagram for *Check campaign budget* use case in [BMF02] is shown in Figure 2.

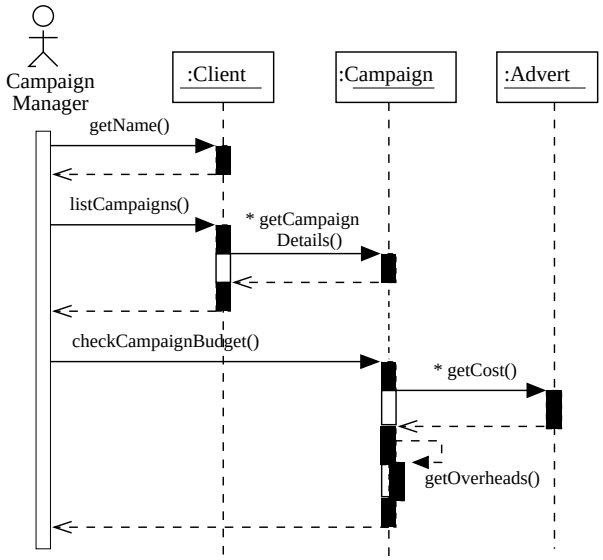

Figure 2: Sequence diagram for *Check campaign budget* showing foci of control and explicit return

The sequence diagram shows the interactions between actor $CampaignManager$ with system three objects which are : $Client$, : $Campaign$ and : $Advert$. There are three system method calls, also called $messages$, which are invoked sequentially by actor $CampaignManager$. They are : $Client.getName()$, : $Client.listCampaign()$, and : $Campaign.checkCampaignBudget()$. In the diagram, we can also find that there are other three messages which are invoked by above three system method calls. The message : $Campaign.getCampaingDetails()$ is directly invoked by object : $Client$, and the other two messages : $Advert.getCost()$ and : $Campaign.getOverheads()$ are invoked by object : $Campaign$.

Before defining the semantics of sequence diagrams, we need to give the formal syntax of sequence diagrams. We introduce an ordered hierarchical structure tree to present well-formed sequence diagram for a thread with synchronous method calls. For the complex concurrent system with multithreads and asynchronous method calls can be defined based on this work.

3.1 Syntax of Sequence Diagram

For a sequence diagram of one thread, there is a corresponding ordered structure tree in which the root node denotes the starting actor or object, tree nodes are objects, and edges represent links and passing messages. Since a sequence diagram is defined as a sequence of messages, we should give formal definition of messages.

Definition 1 A $message$ is a tuple:

$$msg = (ob_i : C_i, ob_j : C_j, action, order)$$

where

- ob_i is the source object of the message with class type C_i, of course the source of message can also be an actor.
- ob_j is the target object of the message with class type C_j.
- $action$ is a guarded method call of the form $g \rightarrow act$, where g is a Boolean expression of attributes of source object ob_i, and possibly public variables, and act is either a command without method calls (an internal action) or a method call $ob_j.m()$.
- $order$ is the order number of the message in the corresponding sequence diagram structure tree. All the order numbers of messages in the sequence diagram construct a partial order. The order number of a message is given according to its position in the tree. The root node is corresponding to the starting object or actor of a sequence diagram. Then the first layer branches with order number is $1,\ 2,\ ,3, \cdots,\ n$. From the u-th node ob_u of first layer object nodes, perhaps there are m nodes. The corresponding order numbers of branches are $u.1,\ u.2, \cdots, u.m$. The $u.v$ is the v-th branches from the node object ob_u. Given the sequence diagram example in Figure 2, the corresponding structure tree with ordered messages is shown in Figure 7.

3.2 Hierarchical Structure Tree of Sequence Diagram

We now look at how composite diagrams can be constructed from basic diagrams by using programming language-like constructs.

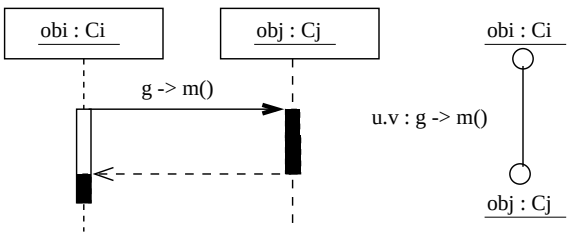

Figure 3: Basic activation notation

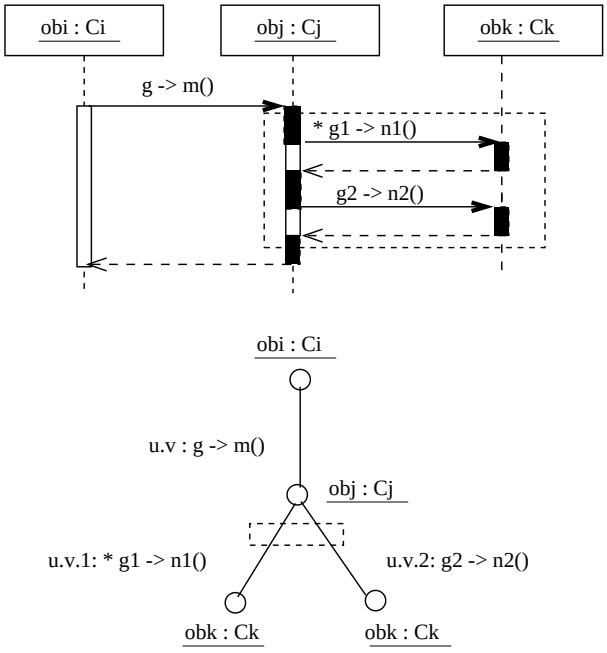

Figure 4: Iteration notation

- The activation notation of a basic message in sequence diagram is shown in Figure 3.
- The iterative notation of messages is shown in Figure 4. The corresponding ordered tree can be constructed in the figure.

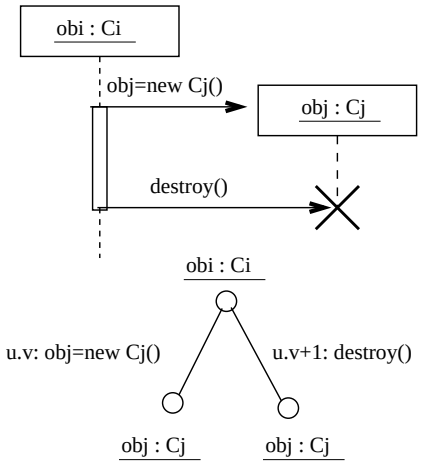

Figure 5: Create and destroy notations

- The special messages such as $newC_j()$ for the creating an object and $destroy$ are shown in Figure 5.
- And the branching choice case of messages can be dealt as shown in Figure 6.

A complicate sequence diagram can be constructed by above basic notations. For sequence diagram of Figure 2, we can easily derive its corresponding message ordered structure tree as shown in Figure 7.

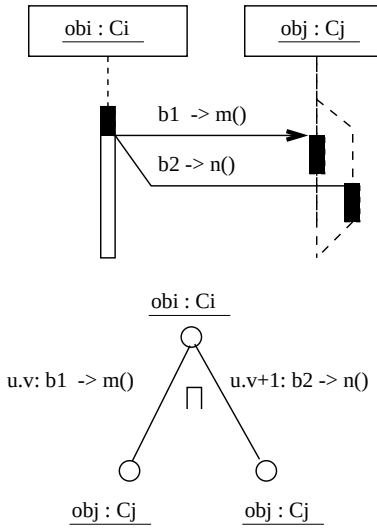

Figure 6: Branching choice notation

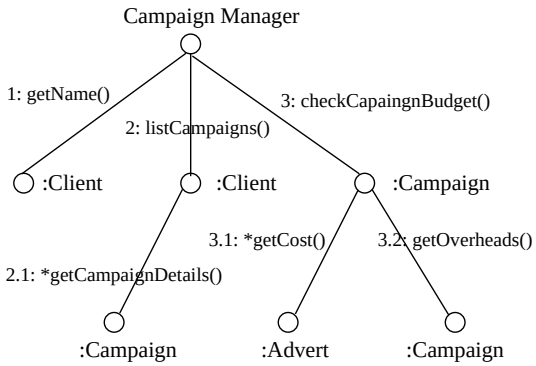

Figure 7: Sequence diagram structure tree

The ordered structure tree presents the hierarchical relationships among the messages. And the execution of messages (i.e. execution of the method represented by the message) in the sequence diagram must follow the traversing rule of first root then son trees from left to right. For one thread sequence diagram, there is only one token to denote control right of invoking message. In the beginning of an execution of the sequence diagram, the token is in the actor's hand. When the source of message wants to invoke a message, the source object or actor must hold the token. When the message is invoked, the token should be passed the target of message. The source of message will wait until the target return the token, it can invoke another message or return the token to the previous source which invoked it before. We consider that the execution of a sequence diagram from the start to termination is the process of the token traversing from the root node of the hierarchical tree to subtrees from left to right, finally the token return back to the root.

We can also use a stack to store the objects for controlling the execution process of the sequence diagram with one thread. At the beginning of execution, the corresponding controlling stack is $empty$. And then the first action is to push the starting object or actor into the stack. The current execution control right is owned by the top element. If the top element object invokes a method call, then the invoked object should be pushed into the stack. After the message returns, the top element object should be popped out. Therefore, the execution of sequence diagram is the process of the nodes of its structure tree entering and leaving the stack. Finally the

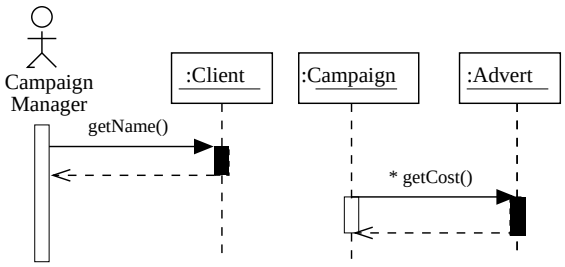

Figure 8: Illegal sequence diagram

execution terminates and the stack becomes empty again. For example, the stack stk of the sequence diagram in Figure 2 is as the following traces

$stk = \varnothing$ //* initial state
$stk =< CM >$ //* start
$stk =<: Client, CM >$ //* execute $getName()$
$stk =< CM >$ //* finish $getName()$
$stk =<: Client, CM >$
$stk =<: Campaign, : Client, CM >$
$stk =<: Client, CM >$
$stk =< CM >$
$stk =<: Campaign, CM >$ //* loop
$stk =<: Advert, : Campaign, CM >$
$stk =<: Campaign, C.M. >$
$stk =<: Advert, : Campaign, CM >$
.....
$stk =<: Campaign, : Campaign, CM >$
$stk =<: Campaign, CM >$
$stk =< CM >$ //* finish $checkCapaingBudget()$
$stk = \varnothing$ //* termination

where CM is the abbreviation of $CampaignManager$.

If the order of the messages given in the sequence diagram with one thread is not conformed to a hierarchical structure tree, then the diagram is not well-formed. For example, the sequence diagram shown in Figure 8 is illegal because after the first message $getName()$ finished, the thread control point is returned the actor $CampaignManager$, however the next message $*getCost()$ in the diagram should be invoked by object $: Campaign$ which does not own the thread control right at that time.

4 Computational Model

We use a notation similar to a transition or action system [MP81] to combine the models of class diagrams and the model of sequence diagrams together to a design model of a system. A *system* is defined by a tuple $(\alpha, \Phi, \Theta, P)$ where

- α denotes the set of program variables known to the program.
- P is a set of *operations*, each of which is a predicate that relates the initial values of program variables. The predicate is of the form $p(x) \vdash R(x, x')$ (called a design in [HH98]):

  $$p(x) \vdash R(x, x') \stackrel{def}{=} ok \wedge p(x) \Rightarrow ok' \wedge R(x, x')$$

  where x and x' represent the initial and final values of x respectively; ok asserting that the operation is started well and ok' means that the operation terminated; $p(x)$ is called the precondition, and $R(x, x')$ the post-condition or the transition relation.
- Θ is a predicate over α, called the *initial condition* and defines the initial state(s) of the system.
- Φ is a predicate over α, called the *invariant*. It must be true in any initial state and preserved by each operation in P.

An action only changes a subset of variables declared in α. The *normal form* of a design is thus a *framed design* of the form $V : (p \vdash R)$, that denotes $p \vdash R \wedge (\underline{w}' = \underline{w})$, where V and $\underline{w}$ are subsets of α, and $\underline{w} = \alpha - V$. When there is no confusion, we will omit the frame in a design by assuming that a variable x can be changed by a design only if its primed version x' occurs in the design.

The above model has to take into account the following OO aspects.

1. A sequence diagram is composed from a number of operations, i.e. method calls, while a class diagrams determines the following variables on which sequence diagram operates:
   - for every class, a *class variable* that takes values of sets of objects of the concept;
   - for every object of a concept, a variable for each attribute of the concept;
   - for every association, an *association variable* that take values of sets of links (i.e. pairs) between objects of the associated classes.
2. Due to the inheritance mechanism, the effect of a use case on a variable depends on its current type during execution, rather than its originally declared type.
3. As in imperative languages, a state of a variable is its current value. An object is represented as a finite tuple that records its *identity*, current type, and the values of its attributes.

In summary, an OO design model is a system $\mathbf{S} = (\alpha, \Phi, \Theta, P)$ where

- P consists of a set of operations obtained from a number of sequence diagrams.
- α identifies the variables on which the operations in P operate and it is determined by the class diagram and the input and output parameters of the methods in the sequence diagram.
- The invariant Φ formally models the invariant constraint, such as multiplicities and business rules. The pair (α, Φ) thus gives the formalization of the class model.
- Θ is a condition to be established when starting up the system.

5 Formal Semantics of Sequence Diagram

We consider both static and dynamic semantics of sequence diagrams. In an abstract syntactic form, a well-formed sequence diagram corresponds to an ordered hierarchical tree structure. The static semantics of a sequence diagram is to check whether it is consistent with the class diagram declaration. This means that the static semantics is always dependent on a given class diagram Δ. Meanwhile, the dynamic semantics is defined as a sequence composition of the first layer system messages of its ordered tree. When a message is executed, it must be consistent with system state, i.e., object diagram and the state diagrams of the relevant objects.

5.1 Static semantics of sequence diagram

Given a class diagram Δ, let CN be the set of class names in Δ, AN the set of association names, Ass the set of all associations each of which is represented in the form $< C_1, A, C_2 >$, where $C_1, C_2 \in CN$ and $A \in AN$, and $method(C)$ be the set of all the methods of class C in Δ. Obviously, for a given message

$$msg = (ob_i : C_i, ob_j : C_j, g \rightarrow m(), order)$$

we can define its static semantics as follows.

$$\mathcal{S}_s[\![smg]\!] \stackrel{def}{=} C_i \in CN \wedge C_j \in CN \wedge \\ \exists A \in AN \cdot < C_i, A, C_j > \in Ass \wedge m \in method(C_j)$$

The static semantics of a sequence diagram is defined as the conjunction of all its messages in the structure tree. Therefore, the *static consistency* of a sequence diagram is captured by this static semantic condition, i.e. a sequence diagram is *statically consistent* (in the context of a class diagram Δ), iff the static semantics of every message is $true$. In particular, all classes of objects in the sequence diagram must be a class in the class diagram, every method from a class C to another class D must be a declared method in class D, and there must be an association between class C and D so that the method of D can be called by C. The checking of this kind of consistency can be easily automated.

However, the static semantics is not enough for the analysis of the dynamic consistency, such as the multiplicity constraints on associations and other state constraints on a class diagram [LLH02, LHLC03] or when two objects are actually *linked* when a method of one is being invoked by the other in a state. This requires a dynamic semantics and we present it in the following section.

5.2 Dynamic Semantics of Message

Execution of a message makes system change from one state s to another s'. Based on Hoare and He's *Unifying Theories of Programming* [HH98], we can define a message action as first order logic formula on a pair of system states (s, s'). Thus, the dynamic behavior of a sequence diagram can be considered as

the sequential composition of executions of its containing messages with some internal small connecting programs. The sequential composition corresponds to relational composition as follows:

$$P(s,s');Q(s,s') \stackrel{def}{=} \exists m \cdot P(s,m) \wedge Q(m,s')$$

Before giving the semantics of messages, we introduce some useful auxiliary functions: for states s and s',

- $s(C)$: the set of objects of class C in state s.
- $s(ob)$: the corresponding state of object ob in system state s, i.e. the values of the attributes of ob.
- $s(g)$: a boolean value of guard condition g under system state s.
- $Enable(s, ob, m)$: a boolean value. It equals $ture$ if the method m of object ob is enabled under system state s.
- $pre_s(ob.m)$: the precondition of method m of object ob under system state s.
- $post_{(s,s')}(ob.m)$: the postcondition of method m of object ob under system state pair (s, s').
- $Link(s, ob)$: the object set which can be navigated from object ob by its associations in the system state s.

we can define the dynamic semantics of action execution (except $create$ action) as follows:

- A method invocation can only be executed when the method is enabled in the current state:

  $$\begin{array}{l}\mathcal{S}_d[\![ob.m()]\!] \stackrel{def}{=} \\ Enable(s, ob, m) \wedge \\ (pre_s(ob.m) \Rightarrow post_{(s,s')}(ob.m))\end{array}$$

  There when $pre_s(ob.m)$ does not hold in a state, the invocation would be $chaos$, and thus the sequence diagram is not *dynamically consistent*.

- A guarded message invocation can go ahead only when the guard is evaluated to be in the current state:

  $$\mathcal{S}_d[\![g \rightarrow ob.m()]\!] \stackrel{def}{=} s(g) \wedge \mathcal{S}_d[\![ob.m()]\!]$$

- Branching provides a choice according to guards or non-deterministically if both guards are true:

  $$\begin{array}{l}\mathcal{S}_d[\![b_1 \rightarrow ob_1.m() \sqcap\ b_2 \rightarrow ob_2.n()]\!] \stackrel{def}{=} \\ s(b_1) \wedge \mathcal{S}_d[\![ob_1.m()]\!] \vee s(b_2) \wedge \mathcal{S}_d[\![ob_2.n()]\!]\end{array}$$

- An iteration repeats the execution of the act, and finally terminates until the guard becomes false:

$$\begin{array}{l}\mathcal{S}_d[\![b * act]\!] \stackrel{def}{=} \\ \exists n \geq 0 \exists s_0, s_1, \cdots, s_n \cdot \\ (s_0(b) \wedge \mathcal{S}_d[\![act]\!]; \\ \ s_1(b) \wedge \mathcal{S}_d[\![act]\!]; \cdots; \\ \ s_{n-1}(b) \wedge \mathcal{S}_d[\![act]\!] \wedge \neg s_n(b))\end{array}$$

  If the iteration never terminates, the semantics of the act can be described as follows:

$$\mathcal{S}_d[\![b * act]\!] \stackrel{def}{=} (s(b) \wedge \mathcal{S}_d[\![act]\!] \wedge s'(b))^*$$

- To destroy an object is to remove the object from the system state:

$$\begin{array}{l}\mathcal{S}_d[\![ob_i.destroy()]\!] \stackrel{def}{=} \\ s'(C_i) = s(C_i) - \{ob_i\} \wedge \\ \forall c \in s(C) \cdot (Link(s', c) = Link(s, c) - \{ob_i\})\end{array}$$

  where C_i is the type class of ob_i, and c is any existing object in current system state.

When a message is executed, some consistent conditions should be checked under system state. The dynamic semantics of a message: $(ob_i : C_i, ob_j : C_j, act)$, can be defined on a pair system (s, s') as follows, where act is not $create$ object action.

Let

$$P \lhd b \rhd Q \stackrel{def}{=} (b \wedge P) \vee (\neg b \wedge Q)$$

and we denote it by

if b **then** P **else** Q

The dynamic semantics of a message can be defined as follows:

$$\begin{array}{l}Poss \stackrel{def}{=} \\ ob_i \in s(C_i) \wedge ob_j \in s(C_j) \wedge ob_j \in Link(s, ob_i)\end{array}$$

$$\begin{array}{l}\mathcal{S}_d[\![(ob_i : C_i, ob_j : C_j, act)]\!] \stackrel{def}{=} \\ \textbf{if } Poss \textbf{ then } \mathcal{S}_d[\![act]\!] \textbf{ else } false\end{array}$$

Therefore, the guard $Poss$ here also ensures that an invocation of a method of an object by another object is not allowed when there is no link between the two objects. This consistency condition is often violated by designers [1].

[1] The paper [L*et al*03] reported 82 percent or more people make mistakes in this aspects.

Now let us deal with the case of $create$ action. An object ob_i can create a new object ob_j only when ob_i has already existed and the new object ob_j must not exist in the current state. The semantics of the action $create$ as $ob_j =$ **new** $C_j()$ is defined as follows:

$$
\begin{array}{l}
\mathcal{S}_d[\![(ob_i : C_i, ob_j : C_j, ob_j = \textbf{new } C_j())]\!] \stackrel{def}{=} \\
\textbf{if } ob_i \in s(C_i) \wedge ob_j \notin s(C_j) \textbf{ then} \\
\quad \exists\, ob \notin s(C_j).(ob'_j = ob) \wedge s'(C_j) = s(C_j) \cup \{ob\} \\
\quad \wedge\ Link(s', ob_i) = Link(s, ob_i) \cup \{ob\} \\
\textbf{else } false
\end{array}
$$

5.3 Dynamic semantics of sequence diagram

The dynamic semantics of a sequence diagram can be defined as the sequential composition of the first layer messages in its corresponding ordered structure tree discussed in Section 3. For example, the message sequence $< msg_1, msg_2, \cdots, msg_n >$ is the first layer messages from left to right of a sequence diagram SD. The semantics of SD can be defined as follows.

$$
\begin{array}{l}
\mathcal{S}_d[\![SD]\!] \stackrel{def}{=} \\
c_1; \mathcal{S}_d[\![msg_1]\!]; c_2; \cdots; c_n; \mathcal{S}_d[\![msg_n]\!]; c_{n+1}
\end{array}
$$

where $c_1, c_2, \cdots, c_n$ and c_{n+1} are small pieces of program or $skip$ defined in the method, which are defined in software implementation phase. If a group of neighbor messages are in choice or iterative relations rather than in sequence relation, we can handle the semantics similarly as the dynamic semantics definition of above message.

For a composite message msg_i with a sequence message $< msg_{i.1}, msg_{i.2}, \cdots, msg_{i.m} >$ at the layer below, its semantics can be roughly defined as follows because some execution information of the method is not provided explicitly in the sequence diagram.

$$
\begin{array}{l}
\mathcal{S}_d[\![msg_i]\!] \stackrel{def}{=} \\
c_1; \mathcal{S}_d[\![msg_{i.1}]\!]; c_2; \cdots; c_m; \mathcal{S}_d[\![msg_{i.m}]\!]; c_{m+1}
\end{array}
$$

However, a sequence diagram is generally invoked by an actor of the system [Lar01]. In this case, the semantics can be simply defined as follows since there are not pieces programs between two system method calls.

$$\mathcal{S}_d[\![SD]\!] \stackrel{def}{=} \mathcal{S}_d[\![msg_1]\!]; \cdots; \mathcal{S}_d[\![msg_n]\!]$$

For example, the dynamic semantics of the sequence diagram in Figure 2 can be defined as follows.

$$
\begin{array}{l}
\mathcal{S}_d[\![SD]\!] \stackrel{def}{=} \mathcal{S}_d[\![: Client.getName()]\!] ; \\
\mathcal{S}_d[\![: Client.listCampaign()]\!] ; \\
\mathcal{S}_d[\![: Campaign.checkCampaignBudget()]\!]
\end{array}
$$

where the two messages : $Client.listCampaing()$ and : $Campaign.checkCampaignBudget()$ must be executed as the following defined execution traces.

$$\mathcal{S}_d[\![: Client.listCampaign()]\!] = c_1;$$
$$\mathcal{S}_d[\![b_1 * : Compaign.getCampaignDetails()]\!]; c_2$$

$$\mathcal{S}_d[\![: Campaign.checkCampaignBudget()]\!] =$$
$$c_3; \mathcal{S}_d[\![b_2 * : Advert.getCost()]\!]; c_4;$$
$$\mathcal{S}_d[\![: Campaign.getOverheads()]\!]; c_5$$

where b_1 and b_2 two omitted Boolean expressions which are omitted in the diagram. And c_1, c_2, c_3, c_4, and c_5 are small programs of internal travail actions shown as five black pieces in the figure. They are not explicitly described in the diagram.

6 Conclusion and Future Work

6.1 Conclusion

The most informal parts of UML are the descriptions of use cases and the links between different UML diagrams. Reports on teaching and using UML for software development show that the majority of inconsistencies are caused by the lack of a precise understanding of these issues[2]. For example, the report [L*et al*03] at the UML 2003 Workshop on Consistency of UML shows that more than 80 percent of students on a project making mistakes in drawing sequence diagrams which contain message passing between unlinked objects. This paper, together with our work presented in [LLH02, LHLC03], address exactly these issues. We define the semantics of a sequence diagram in the context of a class diagram that is also formalized. The formalization is based on a classic computational model of transition systems that are equivalent to UML state diagrams. The restriction of the semantic state transition system on a given object in a sequence diagram gives a state diagram of that object. Therefore, the consistency between a sequence diagram and a given state diagram becomes the consistency between this state diagram and the one obtained from the semantics of the sequence diagram.

Based on this dynamic semantics, we can check whether a sequence diagram realizes a use case whose formal specification was given in [LLH02]. Suppose that SD_1 is a sequence diagram for realizing its use case US with formal specification $p \vdash R$. Therefore, the logical relation between them can be described as follows:

$$(s, s') \vDash (\mathcal{S}_d[\![SD_1]\!] \Rightarrow p \vdash R)$$

6.2 Related work

Related work There is now a large amount of work on formalization of UML, e.g. [Are00, BMF02, BSL01, BGHea98, EFLR98, POB02, Tsi01]. It is not easy to give a full account of comparison. However, there

[2]The Second author attended the UML 2003 Workshop on Consistency of UML. There were three talks, e.g. [L*et al*03], of this kind.

are mainly two kind of publications. The first is the so called *transformational approach* in which certain UML diagrams are translated to an existing formalism, such as Z, B, VDM, CSP, Petri-Nets, PVS , etc. The advantage of this approach is that the tools exist for the well established for reasoning after the translation. The other approach is to directly provide formal semantic models for the UML models and then provide the combination of the different models for consistency checking. Our work belongs to the later. We believe that our work focus on the most informal aspects of UML that are those related to formalization of use cases [LLH02] and the semantic combination of different UML models. We want to ensure that a conceptual class diagram is constructed to support the realization of a certain family of use cases, and a family of interaction diagrams is to describe of the interactions among objects of a class diagram, and a state diagram is only defined in the context of a class diagram. Also, our approach will faithfully preserve the multi-view with multi-notation in UML, that we believe to be its most important advantage compared to a single notational modelling framework such as CSP, Z, VDM, etc., each of which is very natural when dealing with certain aspects of a system, but may not be that nice for other aspects.

We have another paper [LLHL] in this volume. The difference between this paper and that one is that paper focus on formalizing design models which include sequence diagrams in a formal design calculus presented in [HLL02, HLL03]. It is more on the correctness and refinement of a UML design, and intends to support model driven development by means of model transformation. This paper looks at a computational model in details and it focus more on consistency and behavioral analysis.

Our work is not in the area of design algorithms for automatic checking of consistency of UML models or for identifying inconsistencies of UML models, e.g. [BTY03], or development of tools for tool for consistency management in UML-based development, such as [GEK03]. However, the semantics provided in this paper supports formal verification of the correctness of such algorithms, and the development of such a tool.

6.3 Future work

In this paper, we have not considered concurrent execution in the sequence diagram. That means the message passing is synchronous rather than asynchronous. For the asynchronous message passing in sequence diagram, the corresponding semantics can be similarly defined as a concurrent processes of CSP. Meanwhile we can also add the timing constraint mechanism into the sequence diagram. New version of UML2.0 [OMG03] on sequence diagram will adopted in this part of our future work. Future work also includes the formal link between a UML model of requirements and a UML design model. The former model consists of a conceptual class diagram and a use-case model [LLH02], and the later one defined as this paper consists of a class diagram and a family of sequence diagrams.

Acknowledgement: Many thanks to the referees' valuable comments on this paper.

References

[AM02] N. Aguirre and T. Maibaum. A temporal logic approach to component-based system specification and verification. In *Proc. ICSE'02*, 2002.

[Are00] D. B. Aredo. Semantics of uml sequence diagram in pvs. In *online Proc. of UML2000 Workshop on Dynamic Behavior in UML models: Semantic Questions*. York, UK, October 2000.

[BGHea98] R. Breu, R. Grosu, C. Hofmann, and et al. Exemplary and complete object interaction descriptions. In *Computer Standards and Interfaces: 19(1998)*, pages 335–345, 1998.

[BMF02] S. Bennett, S. McRobb, and R. Farmer. *Object-Oriented Systems Analysis and Design using UML, (2nd edition)*. Mc-Graw Hill, 2002.

[BRJ99] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modelling Language User Guide*. Addison-Wesley, 1999.

[BSL01] S. Bennett, J. Skelton, and K. Lunn. *Schaum's Outline of UML*. Mc-Graw Hill, 2001.

[BTY03] B.Litvak, S. Tyszberowicz, and A. Yehudai. Behavioural consistent validation of uml diagrams. In *Proc. 1st IEEE International Conference on Software Engineering and Formal Methods (SEFM)*, pages 118–125. IEEE Computer Society, 2003.

[EFLR98] A. Evants, R. France, K. Lano, and B. Rumpe. Developing the UML as a formal modeling notation. In *Computer Standards and Interfaces: Special Issues on Formal Development Techniques*, 1998.

[GEK03] R. Heckel G. Engels and J. Kuster. The consistency workbench: a tool for consistency management in uml-base development. In J. Whittle P. Stevens and G.Booch, editors, *The Proceedings of UML 2003 in LNCS 2865*, pages 356–359. Springer-Verlag, October 2003.

[HH98] C.A.R. Hoare and J. He. *Unifying theories of programming*. Prentice-Hall International, 1998.

[HLL02] J. He, Z. Liu, and X. Li. Towards a refinement calculus for object-oriented systems. In *Proc. of 1st IEEE International Conference on cognitive Informatics (ICCI02)*, pages 69–76, Alberta, Canada, 2002. IEEE Computer Scociety.

[HLL03] J. He, Z. Liu, and X. Li. Modeling object-oriented programming with reference type and dynamic binding. Technical Report UNU/IIST Report No 279, UNU/IIST, P.O. Box 3058, Macau, May 2003.

[Lar01] C. Larman. *Applying UML and Patterns*. Prentice-Hall International, 2001.

[L*et al*03] C. Lang and *et al*. An impirical investigation in quantifying inconsistency and incompleteness of UML designs. http://www.ipd..pdh..se/consistencyUML/, 2003.

[LHLC03] Z. Liu, J. He, X. Li, and Y. Chen. A relational model for formal object-oriented requirement analysis in uml. In J.S. Dong and J. Woodcock, editors, *Formal Methods and Software Engineering, 5th International Conference on Formal Engineering Methods(ICFEM2003), in LNCS 2885*, pages 640–664, Singapore, 2003. Springer-Verlag.

[LLH02] Z. Liu, X. Li, and J. He. Using transition systems to unify uml models. In Chris George, editor, *The Proceedings of 4th International Conference on Formal Engineering Methods (ICFEM2002), in LNCS 2495*, pages 535–547, Shanghai, China, October 2002. Springer-Verlag.

[LLHL] J. Liu, Z. Liu, J. He, and X. Li. Linking UML models of design and requirement. In *Pro. of Australian Software Engineering Conference (ASWEC'2004)*, Melbourne, Australia. IEEE Computer Sciety.

[MP81] Z. Mana and A. Pnueli. The temporal framework for concurrent programs. In R.S. Boyer and J.S. Moore, editors, *The Correctness Problem in Computer Science*, pages 215–274. Academic Press, 1981.

[OMG03] OMG. Unified Modelling Language Specification, version 2.0. Object Management Group, URL: http://www.omg.org/technology/documents/, 2003.

[POB02] R. Paige, J. Ostroff, and P. Brooke. Checking the consistency of collaboration and class diagram using pvs. In *Proc of Fourth Workshop on Rigorous Object-Oriented Methods(ROOM4), London, England*. British Computer Society, 2002.

[RJB99] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modelling Language Reference Manual*. Addison-Wesley, 1999.

[Tsi01] A. Tsiolakis. Integrating model information in uml sequence diagrams. In *Electronic Notes in Theoretical Computer Science*. URL http://www.elsevier.nl/locate/entcs, 2001.