A Characterization of UML Diagrams and their Consistency *

Xiaoshan Li

Department of Computer and Information Science
Faculty of Science and Technology
University of Macau, Macao, China
Email: xsl@umac.mo

Abstract

In this paper, we focus on giving a formal semantics of sequence diagram of UML2.0. A model of a system comprises a set of program variables, a class diagram, a family of normal sequence diagrams, a family of normal state diagrams, and a state constraint. We then define their static and dynamic consistency supported by the formal semantics.

Keywords: Semantics, UML2.0, Sequence Diagram, Consistency Checking.

1 Introduction

The Unified Modeling Language (UML) [4, 16] 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 UML2.0 sequence diagram only with one thread and synchronous interactions, as well its consistency with other diagrams: class diagram, object diagram and state diagram. The motivation is to provide a natural and intuitive formal semantics of UML2.0 sequence diagram based on the people’s general understanding of object-oriented programming language.

A sequence diagram shows an interaction arranged in time sequence. In particular, it shows the objects participating in the interaction by their lifelines and the messages they exchange, arranged in time sequence. The major improvement in UML2.0 comparing with UML1.4 is on the interaction of sequence diagram. There are some new features introduced in UML2.0 sequence diagram, such as combined fragment, reference interaction use.

In fact, different UML diagrams are used to model system from different views. 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 (communication diagrams), state machine diagrams, and activity diagrams. An object diagram is the instance of a class diagram. As for component and deployment diagrams, they 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 model of design correctly realizes a model of system requirements?

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 machine diagram?
- Does a sequence diagram correctly realize a use case of which formalization was given in [13]?
For the most of software engineers study and use UML for system analysis and design, they may not read formal semantics. The informal semantics in J. Rumbaugh’s UML reference manual book [16] is good enough. However, such a formal semantics is necessary for the development of UML tools, such as consistency checking of system different models and code generation. And this formal semantics is also helpful for system analysts and designers to understand UML diagrams comprehensively.

In this paper, we define a static semantics for UML2.0 sequence diagrams to support checking the well-formedness of a sequence diagram in the context of other diagrams, i.e. its consistency with a class diagram and state machine diagrams; and a dynamic semantics that defines the behavior of a sequence diagram as traces of sending and returning events of method calls.

Since we only consider the synchronous operation call without concurrency, there are only two kinds of call events: sending and returning. The sending and receiving events of method call always occur at the same time, therefore we only talk sending events. The signal, time and change events in asynchronous and concurrent systems are 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 (multithreads) can be considered in an extension of this single thread semantic model.

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 sequence 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 basis that will be used in the rest of the paper. In Section 3, we define class diagrams and object diagrams. Section 4 formalises state diagrams. We devote Section 5 to the study of the static semantics of sequence diagrams. In Section 6, the formal semantics of sequence diagram is given in terms of event traces. Section 7 investigates the consistency between the event trace semantics consistent with object diagrams and state diagrams. Finally, the conclusions are drawn in Section 8 with some discussions on the semantics as well as related and future work.

2 The Semantic Basis

We take a classical approach to modelling the execution of a program in terms of a relation between the states of the program. However, the concept of state is more general than what programmers usually understand and it depends on what the modeler wants to observe of the execution of a program. For example, for a terminating sequential program, we are only interested in the initial inputs and final outputs. For a possible non-terminating program, we need an observable by which we can describe if the program terminates for some inputs. For concurrent and communicating program, we would like to observe the possible traces of interactions, divergencies and refusals, in order to verify if a program is deadlock free and livelock free. If we are interested in real-time programs, we need to observe the time. Identification what to observe in different kinds of systems is one of the core ideas of the Unifying Theories of Programming [8].

2.1 Observables of a program,

We call what to be observed of a program \( P \) the observables or alphabet of the program, denoted by \( \alpha(P) \) and simply \( \alpha \) when there is no confusion. An observable of \( P \) may take different values for different executions or runs, but from the same value space called the type of the observable. Therefore, an observable is also a variable. Though not all observables have to appear in a program text, but they are all needed to define the semantics of the program.

Given an alphabet \( \alpha \), a state of \( \alpha \) is a (well-typed) mapping from \( \alpha \) to the value spaces of the observables. A program \( P \) with an alphabet \( \alpha \) is then defined as a pair of predicates, called design and represented as \( \text{Pre} \vdash \text{Post} \), with free variables in \( \alpha \). It means that if the value of observables satisfies the precondition \( \text{Pre} \) at the beginning of the execution, the execution will generate observables satisfying the postcondition \( \text{Post} \).

2.2 Program as designs

This subsection briefly shows how the basic programming constructs can be defined as designs. For details, we refer the reader to [8].

For an imperative sequential program, we are interested in observing the values of the input variables \( \text{in}a \) and output variables \( \text{out}a \). Here we take the convention that for each input variable \( x \in \text{in}a \), its primed version \( x' \) is in an output variable in \( \text{out}a \), that gives the final value of \( x \) after the execution of the program. We use a Boolean variable \( \text{ok} \) to denote whether a program is started properly and its primed version \( \text{ok}' \) to represent whether the execution has terminated. The alphabet \( \alpha \) is defined as the union \( \text{in}a \cup \text{out}a \cup \{\text{ok}, \text{ok}'\} \), and a design is of the form

\[
(p(x) \vdash R(x, x')) \overset{\text{def}}{=} \text{ok} \land p(x) \Rightarrow \text{ok}' \land R(x, x')
\]

where

- \( p \) is a predicate over \( \text{in}a \) and \( R \) is a predicate over \( \text{out}a \),
• \( p \) is the \textit{precondition}, defining the initial states
• \( R \) is the \textit{postcondition}, relating the initial states to the final states.
• \( ok \) and \( ok! \): describe the termination, they do not appear in expressions or assignments of program texts

The design represents a \textit{contract} between the “user” and the program such that if the program is started properly in a state satisfying the precondition it will terminate in a state satisfying the postcondition \( R \).

A design is often \textit{framed} in the form

\[
\beta : (p \vdash R) \overset{def}{=} p \vdash (R \land w' = w)
\]

where \( w \) contains all the variables in \( ino \) but those in \( \beta \).

Before we define the semantics of a program, we first define the some operations on designs.

• Given two designs such that the output alphabet of \( P \) is the same as primed version of the input alphabet of \( Q \), the sequential composition

\[
P(ino_1, out_1); \ Q(ino_2, out_2) \overset{def}{=} \exists m : P(ino_1, m) \land Q(m, out_2)
\]

• Conditional choice:

\[
(D_1 \triangleleft b \triangleright D_2) \overset{def}{=} (b \land D_1) \lor (\neg b \land D_2)
\]

This can also be denoted by if \( b \) then \( D_1 \) else \( D_2 \).

• \textbf{while} \( b \) do \( D \) is defined as the worst fixed point

\[
X = ((D; X) \triangleleft b \triangleright skip)
\]

We can now define the meaning of primitive commands program commands as framed designs as follows. Composite statements are then defined by the operations on designs.

\[
skip \overset{def}{=} \{ \} : \text{true} \vdash \text{true}
\]

\[
\text{chaos} \overset{def}{=} \{ \} : \text{false} \vdash \text{true}
\]

\[
x := e \overset{def}{=} \{ x \} : \text{true} \vdash x' = \text{val}(e)
\]

\[
m(e; v) \overset{def}{=} [\text{var in, out}]; [\text{in}:=e]; [\text{body}(m)]; [\text{v}:=\text{out}]; [\text{end} \text{ in, out}]
\]

where \( skip \) means that it does not change anything, but terminates; \( \text{chaos} \) means that anything, including non-terminating, can happen; \( x := e \) means that side-effect free assignment updates \( x \) with the value of \( e \), and \( m(\text{in, out}) \) is the signature with input parameters \( \text{in} \) and output parameters \( \text{out} \); \( \text{body}(m) \) is the body command of the procedure/method

In general, when defining a particular programming language, the preconditions are usually strengthened with some \textit{well-definedness} conditions of the commands, and a program or command \( e \) is generally of the form

\[
[e] \overset{def}{=} D(e) \Rightarrow \text{Spec}
\]

where \text{Spec} is a design. Some of the well definedness may even be dynamic.

Strengthening precondition with well-definedness conditions allows us to treat correcting a unwell-defined command to a well-formed one as refinement. This is essential to support incremental and iterative development as most cases of unwell-defined are due to the insufflate of data or services. Therefore, adding more data, services and components, without altering the existing ones, will be refinement in our framework.

Within such an approach, we defined a semantics and a calculus, called \text{RyOS}, for object-oriented programs [7]. There, object creation and method invocation are defined as designs too. The model also captures inheritance, subtyping, dynamic binding, and type casting.

2.3 UML model of system

A system model \( \mathcal{M} = \langle \alpha, \Gamma, \Delta, \Omega, \Theta \rangle \) consists of

• a non-empty finite set \( \alpha = \{ x_1 : T_1, \ldots, x_n : T_n \} \) of program variables, class diagram \( \Gamma \),
• a family \( \Delta \) of sequence diagrams,
• a family \( \Omega \) of state machines and
• a system constraint \( \Theta \).

A consistency condition requires that the type \( T_i \) of each variable is either a built-in primitive data type or a class name in \( \Gamma \).

We require that each sequence diagram represents a realization of the interactions of use case and thus starts with an \textit{actor object}. The actor sends a number of method calls in sequence to other objects. Alternatives and loops are allowed. Similarly, each state diagram represent a realization of the functional behaviour of a use case.

3 Class Diagrams and Object Diagrams

The class diagram of a system describes the static structure of a system and defines the state space of the system. It thus defines the environment in which the sequence diagrams and state diagrams behave.

\textbf{Definition 1} A \textbf{class diagram} of an application, \( \Gamma = (\text{CN}, \text{AN}, \text{super}, \text{attr}, \text{meth}) \), identifies the following modelling elements:
1. **CN**: the finite set of class names that captures the concepts of the application domain or software system in the diagram. We use capital letters such as \( C \) and \( D \) to represent arbitrary classes.

2. **super**: the partial function which maps a class to its direct superclass, i.e. \( \text{super}(C) = D \) if \( D \) is the direct superclass of \( C \).

3. **AN** is the set of association names captured in the diagram. \( \text{Ass} \) is a set of triples \( \langle C, A, D \rangle \) called associations names where \( C, D \in \text{CN} \) and \( A \in \text{AN} \) is an association name. For simplicity, we only deal with binary associations. General relations among classes can be modelled in the same way.

4. Each \( C \in \text{CN} \) is associated with two functions \( \text{attr}(C) \) and \( \text{meth}(C) \). \( \text{attr}(C) \) maps \( C \) to the set of \( \{< a_1 : T_1 >, \ldots , < a_m : T_m >\} \) of attributes, where \( T_i \) stands for the type of attribute \( a_i \).

5. \( \text{meth}(C) \) maps \( C \) to a set of method signatures, noted as \( \text{meth}(C) = \{m_1(x_1 : T_{11}; y_1 : T_{12}), \ldots , m_k(x_k : T_{k1}; y_k : T_{k2})\} \), where \( x_i : T_{ij} \) and \( y_i : T_{ij} \) are the value parameter and result parameter of method \( m_i() \). Here, for simplicity but without losing any generality, we assume each method has one value parameter and one result parameter.

The multiplicities of the roles of an association has be specified in the invariant of the system [13] which are not essential to this paper.

**Definition 2** A class diagram \( \Gamma \) is **well-formed** under the following conditions hold:

1. each class name \( M \in \text{cname} \) and the name of its direct superclass \( N \) are distinct,
2. if \( M \in \text{CN} \) and \( \text{super}(M) = N \), the \( N \in \text{CN} \),
3. any type used in declarations of attributes and parameters is either a primitive built-in type or a class in \( \text{CN} \),
4. the transitive closure \( \succ \) of the direct generalization relation \( \text{super} \) is acyclic,
5. any attribute of a class is not redeclared in its subclasses, i.e. we do not allow attribute hiding in a subclass,
6. the names of the attributes of each class are distinct,
7. the names of the methods of each class and the names of parameters of each method are distinct respectively.

An object is an instance of a class. In a snapshot system time point, an object has an unique identity and a state which assigns a value to each attribute of its class. However, a full representation of an object also includes its relation with other objects via associations. This will form an **object diagram**.

**Definition 3** Given a class diagram \( \Gamma \), an **object diagram** \( \Delta \) of \( \Gamma \) is an instance of \( \Gamma \), consisting of the following model elements:

- \( \mathcal{O} \): a set of objects \( \{o_1 : C_1, \ldots , o_n : C_n\} \), where \( o_i \) are the identities and \( C_i \) are class names in \( \text{CN} \), denoted \( \text{type}(o_i) \). We all use \( o_i, a \) to denote the value of attribute \( a \) of object \( o_i \).
- \( \text{Link} \): a subset of \( \mathcal{O} \times \text{AN} \times \mathcal{O} \) such that \( (o_1 : C_1, A, o_2 : C_2) \in \text{Link} \) only if when there is an association \( A \) between class \( C_1 \) and class \( C_2 \) in \( \Gamma \), i.e. \( (C_1, A, C_2) \in \text{Ass} \).

For a system model \( M = \langle \alpha, \Gamma, \Delta, \Omega, \theta \rangle \), a system state \( s \) maps each variable with a primitive to a value in the data type, associates each variable with call type an object of that class, and assigns a truth value (i.e. in \{true, false\}) to the predicate \( \text{link}(x, A, y) \) for each \( (\text{type}(x), A, \text{type}(y)) \in \text{Ass} \). Therefore, each state \( s \) corresponds an object diagram \( \Delta \) of \( \Gamma \). A **valid state** of \( M \) is a state that satisfies the state constraint \( \theta \). Later when we mention a state, we only mean a valid state. Therefore, the validity of a state ensures the consistency among the class diagram, the program variables and the state constraint. For each state \( s \) and a class \( C \), we use \( s[C] \) to denote the set of the objects of class \( C \) in \( s \).

**4 State Diagrams**

In this paper, we use a state diagram to describe the dynamic behavior of a use case realization at a certain level of abstraction. One can understand such a state diagram as a state diagram of a certain object, i.e. the use-case controller object [12]. However, the behaviour of this object involves the behaviour of its associated objects, and their associated objects, and so on. When this object is created, its attributes are initiated and it enters to initial state. Then afterwards the object receives events from outside and does the corresponding actions and makes its state transitions. Finally it may be destroyed by enter final state, i.e. it will be removed from system by memory collection mechanism by computer systems.

**Definition 4** A **state machine diagram** is a 5-tuple \( \text{StD} = \langle \text{Loc}, \text{Event}, L_0, L_f, R \rangle \), where

1. \( \text{Loc} \) is a set of finite locations (i.e. control states). At anytime, the object can only be at one location. \( L_0 \) is
the initial location. And \( L_f \) is the set of final states. When one of its final states is reached, the object will stay in the state until the object is destroyed.

2. Event is the set of events which the object can receive from outside. Generally, an event is a method invocation \( o.m() \), where \( o \) is the object receiving the invocation.

3. \( R \) is the transition function

\[
R \subseteq \text{Loc} \times \text{Event} \times \text{Guard} \times \text{Action} \times \text{Loc}
\]

where an \( \text{action} \in \text{Action} \) is a program statement written in rCOS, which is similar to a Java command but we allow specification statements.

The transition relation \( R \) defines the dynamic behavior of the object. The general form of transition can be denoted as \( \ell \{ e \ [g] \ / \ \text{action} \} \ell' \), which means that when event \( e \) triggers the object state machine, and does the action which makes the object transfer from source state \( \ell \) to target \( \ell' \) under the guard condition \( g \) holds at the event occurrence. If the guard \( g \) does not hold, the transition will not be fired and the object will stay its original state without change, and ignore the event occurrence.

A guarded triggering event \( g, o.m() \) is enabled in a state \( s \) if \( s[o] \) is true, denoted as \( \text{Enabled}(s, o.m()) \).

The state machine diagram also includes information of entry and exit action when transition is fired, substate, and other complex notations. Here we just use the simplified definition.

4.1 Semantics of a state diagram

To define the full semantics of a state diagram \( \text{StD} \), we associate each location \( \ell \) with a set \( \text{state}(\ell) \) of system state and define the functionality of the actions. We use a set of states so that we can deal with refinement of a state diagram in future. The function \( \text{state} \) satisfies the following conditions:

1. \( \text{state}(L_0) \) is the set of possible initial states of state of the object of the state diagram, which defines the initial values of the attributes and the objects that are initially linked via associations. We require that the class diagram provide these details.

2. For each transition \( \ell \{ e \ [g] \ / \ \text{action} \} \ell' \),

\[
\text{state}(\ell') \triangleq \{ s' \mid (s, s') \models g \land \llbracket \text{action} \rrbracket, s \in \text{state}(\ell) \}
\]

where \( \llbracket \text{action} \rrbracket \) is semantics of the command \( \text{action} \) defined in rCOS [7].

5 Static Semantics of Sequence Diagrams

Sequence diagrams are used to present the dynamic behavior of system design while class diagrams are system static structure. A sequence diagram shows interactions between objects arranged in a time sequence. For example, a sequence diagram named \textit{example} is shown in Figure 1, in which message \( m_2 \) and \( m_8 \) are \textit{create} and \textit{destroy} operations. The main mechanism we will discuss in this paper are combined fragments \textit{alt}, \textit{loop} and \textit{ref} all shown in Figure 1.

![Figure 1. Example of a Sequence Diagram](image-url)
SequenceDiagram ::= 
  sname def= CombinedFragment 
CombinedFragment ::= Interaction | 
  CombinedFragment; CombinedFragment | 
  opt(Cond, CombinedFragment) | 
  alt(Cond, CombinedFragment, CombinedFragment) | 
  loop(Cond, CombinedFragment) | 
Interaction ::= skip | ref(sname) | Message 
  | Message def= { (CombinedFragment)} 
Message ::= (Sender, A, Receiver, MethodCall) 
Cond ::= booleanexpression 
Sender ::= objectname : CN 
Receiver ::= objectname : CN 
CN ::= classname 
A ::= associationname 
MethodCall ::= method(para) 

where def expresses the left side message method call will 
invoke the messages inside of the brace body {}.

Definition 5 A message is a tuple: msg = (c : C, A, d : D, m(para)), where
1. c is the sender object of the message with class type C, 
of course the sender of message can also be an actor.
2. d is the receiver object of the message with class type D. Sometimes, d can be the referenced sequence dia-
gram, for interaction use, e.g., message m6 in Figure 1 calls sequence diagram with name use-sd.
3. A is an association between classes C and D for object 
c navigating to reference of object d.
4. m(para) is a method call from sender c to receiver 
object d. If c is the same as d, it is a self nest method call. para is parameter. We ignore it in this paper for 
simplicity, and denote method call as m().

Definition 6 A sequence diagram ∆ = (ObjectSet, 
MessageSet), in which
1. ObjectSet is the set of objects which participate in the 
sequence diagram.
2. MessageSet consists of all the messages in the di-
gram, which are numbered as a sequencing tree. It 
demonstrates the possible execution order as well as the 
implementation relationships among messages.

5.2 Static semantics of sequence diagram

From the definition of class diagram in section 2, a given 
class diagram Γ consists of four parts, in which CN is the 
set of class names, AN the set of association names, the 
association Ass is the set of triple association name A to 

its classes: \(< C, A, D >\), where \(C, D \in CN\) and \(A \in AN\), and \(meth(D)\) be the set of all the methods of class 
\(D\). Obviously, for a given message \(msg = (c : C, A, d : D, m())\), we can define its static semantics as follows.

\[
M_s[smg] \overset{def}{=} C \in CN \land D \in CN \land A \in AN 
\land m() \in meth(D) \land < C, A, D > \in Ass
\]

Although all the messages of a sequence diagram are 
well-formed, we cannot guarantee the sequence diagram 
to be well-formed as well because the messages must be 
connected properly. For example, in Figure 1 the receiver 
object \(b : B\) of method call must be the sender object of 
method call \(m4\) and \(m5\) since they are directly invoked by 
method call \(m3\). In fact, for the sequence diagram with 
one thread and synchronous interactions, the numbers of its 
messages constructs a tree, shown in Figure 2 for sequence 
diagram example in Figure 1.

Many UML books [16, 10, 2] present the message la-
belling method, called sequence-expression. If a sequence 
diagram with one thread only has synchronous message 
calls, the order numbers of all messages construct a mes-
sage number sequencing tree. On the contrary, for a given 
sequence diagram, the relationships of invocation messages 
also decide the tree. The order number of a message can 
also be given according to its position in the tree. The root 
ode is corresponding to the starting object or actor of a se-
quencc diagram. Then the first layer branches with order 
number is \(1, 2, 3, \cdots, n\), which are sequencing method 
calls which are directly invoked by the starting object or 
actor. In communication diagram, people can only use the 
message numbers to identify the timing order and relation-
a method to construct the message number sequencing tree from message invocation relations of a sequence diagram. For example, if a message $msg_i$ with number $n.m$ (both $n$ and $m$ are natural numbers), directly invokes a sequence messages $msg_1, msg_2, \ldots$, and $msg_k$, then the numbers of these $k$ messages are $n.m.1, n.m.2, \ldots$, and $n.m.k$, respectively.

A well-formed sequence diagram, i.e., its static semantics, can be defined as the conjunction of all above static semantics of its messages as well as its numbers of messages can construct a message number sequencing 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 $\Gamma$), iff the static semantics of every message is true. In particular, all classes of objects in the sequence diagram must be declared 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. For the interaction reference ref, it can be checked as a sequence diagram itself first, and then consider it as a leaf in the tree.

6 Dynamic Semantics of Sequence Diagrams

The dynamic semantics of a sequence diagram is naturally described as the all possible observable event traces while its execution.

Each synchronous interaction between two objects corresponds to a pair of call events send and return. For a given message $msg = (c : C, A, d : D, m())$, we denote two events as $send(c, d, g, m)$ and $return(d, c, m)$, where $g$ is the condition for sending method call event for alternative condition combined fragment, and its default means true. Thus, the dynamic semantics of a sequence diagram can be defined as all the possible sequencing traces of the send and return events of all the messages in the diagram. As for interaction use ref, it can be handled as a basic message.

The message number sequencing tree presents the hierarchical relationships among the messages. And the execution of messages in the sequence diagram must follow the traversing rule of first root then son trees from left to right. By traversing the tree, we can easily get the event traces. For example, we can get the following event traces for the semantics of $M_{tr}[\text{example}]$ from the tree of Figure 2.

$$M_{tr}[\text{example}] \overset{\text{def}}{=} send(-, a, -, m1) \land send(a, b, -, m2) \land send(a, b, g, m3) \land send(b, c, -, m4) \land return(c, b, m4) \land return(c, b, m5) \land return(d, b, m6) \land send(a, used-sd, \neg g, m6) \land return(used-sd, a, m6) \land send(a, c, -, m7) \land return(a, c, m7) \land send(a, b, -, m8) \land return(b, a, m8) \land return(a, -, m1)$$

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 also construct the traces of a sequence diagram according to the abstract syntax productions. The notation $M_{tr}[msg]$ is to describe the possible traces whose events appear in the execution of $msg$. If a sequence diagram $sd$ is defined as a combined fragment $Conf$, then $M_{tr}[sd] = M_{tr}[Conf]$.

The dynamic event trace semantics of sequence composition two combined fragments $Conf_1; Conf_2$ as follows.

$$M_{tr}[Conf_1; Conf_2] = M_{tr}[Conf_1] \land M_{tr}[Conf_2]$$

Similarly, we have

$$M_{tr}[\text{opt}(g, Conf)] = send(c, d, g \land g', m)^\ast tail(M_{tr}[[\text{Conf}]])$$

$$M_{tr}[\text{alt}(g, Conf_1, Conf_2)] = send(c, d, g \land g', m_1)^\ast tail(M_{tr}[[\text{Conf}_1]]) \lor send(c, d, \neg g, g_2, m_2)^\ast tail(M_{tr}[[\text{Conf}_2]])$$

$$M_{tr}[\text{loop}(g, Conf)] = \exists n.(send(c, d, g \land g', m)^\ast tail(M_{tr}[[\text{Conf}]]))^n \land send(c, d, \neg g, g', m)^\ast tail(M_{tr}[[\text{Conf}]])$$

$$M_{tr}[\text{loop}(1, n, Conf)] = (M_{tr}[[\text{Conf}]])^n$$

$$M_{tr}[\text{skip}] = ()$$

$$M_{tr}[\text{ref}(sd)] = M_{tr}[sd]$$

$$M_{tr}[[c : C, A, d : D, m()]] = send(c, d, \neg m)^\ast return(d, c, m)$$

$$M_{tr}[[c : C, A, d : D, m()]] = \{Conf\} = send(c, d, \neg m)^\ast M_{tr}[[Conf]] \land return(d, c, m)$$
where we use to operations **head** and **tail** on sequence, and suppose that \( \text{head}(\text{Multi}[\text{Com}]) = \text{send}(c,d,g',m) \), similarly for fragments \( \text{Com}_1 \) and \( \text{Com}_2 \).

## 7 Dynamic Consistency

Execution of a message makes system change from one state \( s \) to another \( s' \). The execution of a sequence diagram can be considered as a sequence of executions of its messages, we can define the logical semantics of message execution as pre and post conditions of the method invocation. This semantics can help us to define the dynamic consistency between sequence diagram with state machine diagrams and system object diagram.

Before giving the logic semantics of messages, we introduce some useful auxiliary functions: for system states \( s \) and \( s' \).

![Figure 3. Dynamic Semantics of Message](image)

When a message is executed, some consistent conditions should be checked under system state. The dynamic semantics of a message: \( \langle c : C, d : D, m() \rangle \), can be defined on a pair system \( (s, s') \) as follows, where \( m() \) is not create or destroy method.

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

\[
\text{Poss} \overset{\text{def}}{=} c \in s(C) \land d \in s(D) \land d \in \text{Link}(s, c, A)
\]

\[
\text{Multi}[\langle c : C, d : D, m() \rangle] \overset{\text{def}}{=} \text{if Poss then}
\quad \text{Enable}(s, d, m) \land (\text{pre}_a(d, m) \Rightarrow \text{post}(s, s')(d, m))
\quad \text{else false}
\]

Therefore, the guard \( \text{Poss} \) 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.\(^1\) It is also called the dynamic consistency between sequence diagram with object diagram, shown in Figure 3.(a) and 3.(b).

On the other hand, the message call should also be consistency with state diagram, shown in Figure 3.(a) and 3.(c). That is, the transition should be enabled under system state, i.e. \( \text{Enable}(s, d, m) \) must be \( \text{true} \). At this situation, if the \( \text{pre}_a(\text{ob, m}) \) holds, the method invocation can be successfully executed and object \( d \) changes its state to \( s' \), and makes \( \text{post}(s, s')(d, m) \) to be \( \text{true} \). If \( \text{pre}_a(\text{ob, m}) \) does not hold in a state \( s \), the invocation would be \( \text{chaos} \), and thus the sequence diagram is not \( \text{dynamically consistent} \).

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

\[
\text{Multi}[\langle c : C, A, d : D, \text{destroy} \rangle] \overset{\text{def}}{=} \text{if Poss then}
\quad (s'(D) = s(D) - \{d\}) \land \forall e : E.(e \in s(E) \land \\
\text{Link}(s', e, A) = \text{Link}(s, e, A) - \{d\}) \text{ else false}
\]

where \( e \) is any existing object with class type \( E \) in current system state \( s \).

Now let us deal with the case of \( \text{create} \) action. An object \( c \) can create a new object \( d \) only when \( c \) has already existed and the new object \( d \) must not exist in the current state. The semantics of the action \( \text{create} \) as \( d = \text{new} \ D() \) is defined as follows:

\[
\text{Multi}[\langle c : C, A, d : D, d = \text{new} \ D() \rangle] \overset{\text{def}}{=} \text{if } c \in s(C) \land d \not\in s(D) \text{ then}
\quad \exists \text{ob} \not\in s(D). (d' = \text{ob}) \land s'(D) = s(D) \cup \{\text{ob}\} \land \\
\text{Link}(s', c, A) = \text{Link}(s, c, A) \cup \{\text{ob}\} \text{ else false}
\]

Therefore, the dynamic semantics of combined fragments can be constructed by basic message interactions.

- Alternative conditional fragment can be defined as follows.

\[
\text{Multi}[\langle g, \text{action}_1, \text{action}_2 \rangle] \overset{\text{def}}{=} \text{if } g \text{ then Multi}[\text{action}_1] \text{ else Multi}[\text{action}_2]
\]

Similarly, combined fragment \( \text{opt} \) can be considered as the special case of \( \text{alt} \) by using \( \text{skip} \) as else interaction, whose semantics \( \text{Multi}[\text{skip}] = \text{false} \).

- An iteration repeats the execution of the \( \text{action} \), and finally terminates until the guard becomes \( \text{false} \).

\[
\text{Multi}[\langle g, \text{action} \rangle] \overset{\text{def}}{=} \exists n \geq 1 \exists s_0, s_1, \ldots, s_n.
\quad (s_0(g) \land \text{Multi}[\text{action}]; \ldots; s_{n-1}(g) \land \text{Multi}[\text{action}] \land \neg s_n(g))
\]

\(^1\)The paper [9] reported 82 percent or more people make mistakes in this aspects.
The logical semantics of a sequence diagram can be defined as the sequential composition of the first layer messages in its corresponding message number sequencing tree discussed in Section 4. For example, the message sequence \(< \text{msg}_1, \text{msg}_2, \cdots, \text{msg}_n >\) is the first layer messages from left to right of a sequence diagram \(SD\). The semantics of \(SD\) can be defined as follows.

\[
M_l[\text{SD}] \overset{\text{def}}{=} M_l[\text{msg}_1]; \cdots; M_l[\text{msg}_n]
\]

If a group of neighbor messages are in conditional choice or iterative relations rather than in sequence relation, we can handle similarly as the above logical semantics definition of combined fragments.

For a composite message \(\text{msg}\), with a sequence of messages \(< \text{msg}_{1,1}, \cdots, \text{msg}_{1,m} >\) at the layer below in the tree, which it directly invokes, its semantics can be defined as follows. It also means that source code of method in \(\text{msg}\) include the sequence composition of methods from \(\text{msg}_{1,1}\) to \(\text{msg}_{1,m}\). This can be used for code generation [14].

\[
(M_l[\text{msg}_{1,1}]; M_l[\text{msg}_{1,2}]; \cdots; M_l[\text{msg}_{1,m}])
\]

This means that the sequence of messages is one kind of executions of the message \(\text{msg}_{1}\).

Therefore, the dynamic semantics of the sequence diagram in Figure 1 with message number sequencing tree in Figure 2, can be defined as follows.

\[
M_l[\text{example}] \overset{\text{def}}{=} M_l[\text{m1}()]\]

where the relations of messages in the sequence diagram can be described as follows:

\[
(M_l[\text{m2}()]; M_l[\text{alt}(y, \text{m3}(), \text{m6}())]; M_l[\text{m7}()]; M_l[\text{m8}()]) \Rightarrow M_l[\text{m1}()]
\]

\[
(M_l[\text{m4}()]; M_l[\text{m5}()]^n) \Rightarrow M_l[\text{m3}()]
\]

\[
M_l[\text{m6}()] = M_l[\text{used-sd}]
\]

8 Conclusion and Future Work

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. For example, the report [9] 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 extends the work of [11] from UML1.4 to UML2.0, and gives the trace semantics of sequence diagram additionally. This paper, together with our work presented in [13, 12], 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 logical semantics, we can check whether a sequence diagram realizes a use case whose formal specification was given in [13]. Suppose that \(SD\) 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:

\[
(s, s') \models \{M_l[\text{SD}] \Rightarrow p \vdash R\}
\]

There is now a large amount of work on the formalization of UML, e.g. [1, 2, 5, 15, 17]. It is not easy to give a full account of comparison. However, there 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 [13] 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.

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. [3], or development of tools for tool for consistency management in UML-based development, such as [6]. However, the semantics provided in this paper supports formal verification
of the correctness of such algorithms, and the development of such a tool.

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. Future work also includes the formal link between a UML model of requirements and a UML model of design. The former model consists of a conceptual class diagram and a use-case model [13], and the later one defined as this paper consists of a class diagram and a family of sequence diagrams.

Acknowledgement: Many thanks to Dr. Zhiming Liu, Prof. Jifeng He, and referees’ valuable comments.

References


