MECHANIZED SEMANTICS OF UML SEQUENCE DIAGRAMS

Ying Zuo
Department of Computer Science and Technology
East China Normal University
Shanghai 200241, China
email: zyzuoying@gmail.com

Liang Dou, Lihua Xu and Zongyuan Yang
Department of Computer Science and Technology
East China Normal University
Shanghai 200241, China
email: {ldou,lhxu,yzyuan}@cs.ecnu.edu.cn

ABSTRACT
UML Sequence Diagrams are widely used in software development. When putting to applications such as code generation, model simulation and other automated analysis, the formalization of UML Sequence Diagrams, especially correctness of it becomes increasingly important. This article presents the formal specification including denotational semantics and operational semantics of UML Sequence Diagrams. The Coq proof assistant is used to mechanize the semantics and prove the correctness of operational semantics compared to denotational semantics.

KEY WORDS
Sequence diagram, Formalization, Mechanized semantics

1 Introduction

UML sequence diagrams (SDs) have long been used to help discussing and designing the system in the early stage of software development process. A sequence diagram specifies runtime behaviors of a system in a graphical manner. Different objects or processes are represented by parallel vertical lines in a sequence diagram. Objects or processes communicate with each other via messages that are represented by horizontal arrows.[13],[16]. Sequence diagrams help designers understand the system behavior with visual-based graphical representations. Software faults can be discovered and removed in design phase by reasoning about properties of sequence diagram design models, thereby improving quality of software and reducing development cost.

Despite of their wide usage, a number of serious problems have been identified, most notably in dynamic views [8]. UML is a semi-formal language, with a precisely defined syntax and static semantics but with an only informally specified dynamic semantics [11]. It is not just an academic desideratum, but very down-to-earth practical problem. Formal semantics are necessary to take full advantage of UML. Furthermore, this forms the first step of relevant works on SD such as code generation, model simulation, etc. Any mistakes in formal semantics may cause critical error in relevant works. Correct formal semantics is important as it can promote the development of UML-based software development, such as refinement and verification of models.

Many researchers have defined formal semantics for SDs with ordinary mathematics [1],[5],[6] , but the correctness of the proposed semantics is still a question. Traditionally, researchers present on-paper proofs to verify the correctness of the semantic. However, with on-paper proofs, the correctness and soundness of the proofs themselves cannot be guaranteed. We believe machine assistance such as the use of proof assistants is a good approach to solve this problem, which could ultimately facilitate the definition and usage of formalization. Mechanic proof has been successfully used to formalize the semantics of programming languages [19]. In this paper, we show mechanized semantics as formal specification of SDs and verify the semantic preservation mechanically.

We explore the idea of using the theorem proof assistant - Coq to mechanize the denotational and operational semantics for a subset of sequence diagrams. The abstract syntax and denotational semantics come from STAIRS [9]. In particular, we prove the correctness (semantic preservation) of operational semantics in three steps: firstly, the equivalence between denotational semantic and their corresponding big-step operational semantics are proved; secondly, the consistency between big-step and small-step semantics are checked and verified; thirdly, the operational semantics are compared to their corresponding denotational semantics for final verification. The semantic preservation property of SD is defined as: forall the traces in denotational semantics, they can be achieved by operational semantics; forall the traces which can be achieved by operational semantics, they can also be achieved by denotational function. To the best of our knowledge, this is the first attempt to define mechanized operational semantics of SDs and prove the correctness mechanically.

The rest of the paper is organized as follows. Section 2 defines the denotational semantics of SDs and section 3 defines the operational semantics. Some desired properties are proved in section 4. Section 5 gives an example to present the formal semantics of an SD. Section 6 introduces related work. Conclusions and future work are discussed in Section 7.

Availability The Coq development underlying this article can be consulted on-line at http://github.com/LisaPhoebeCoq/Formalization-of-sequence-diagram/downloads
2 Denotational semantics

Since denotational semantics is considered as specification of SD, the authority of it has a direct impact on the significance of this work. We use an existing denotational semantics named STAIRS instead of giving a new one. STAIRS is an approach to the compositional development of UML interactions supporting the specification of mandatory as well as potential behavior. It defines a denotational, trace-based semantics for SD. We adopt and simplify the semantic model of STAIRS. Based on the semantic model of STAIRS, this paper formalizes five basic operators of which is formalized. Negative and global operators are not taken into consideration since the denotational semantics only concerns positive traces.

2.1 Abstract syntax

An SD is structured as events and operators. Event is the atom of an SD. It consists of a message and its kind. Let $M$ be the set of message and $K = \{?,!\}$ be the set of event kinds, where $!$ represents transmitting and $?$ represents receiving. An event $e$ is defined as follow

$$e = (k,m) \in K \times M$$

A message includes a signal which represents the content of a message, the address of transmitter and receiver. Let $L$ be the set of all lifelines in the SD. $S$ denotes the signals of diagram which contain the context of message. A message $m$ is defined as a triple:

$$m = (s, t, r) \in S \times L \times L$$

In addition to events, an SD is also constructed by some operators. The UML standard defines eight unary operators $\text{opt}$, $\text{break}$, $\text{loop}$, $\text{critical}$, $\text{neg}$, $\text{assert}$, $\text{ignore}$, $\text{consider}$ and five binary operators $\text{seq}$, $\text{par}$, $\text{alt}$, $\text{loop}$ and $\text{strict}$. As a first step to demonstrate our idea, we handle the unary $\text{opt}$ and the binary $\text{alt}$, $\text{strict}$, $\text{loop}$, $\text{par}$, $\text{skip}$ represents the empty SD.

Let $D$ be the SD and $E$ be the set of events. $D$ is defined recursively as the least set such that:

$$\begin{align*}
\text{skip} & \in D \\
(\forall e \in E \Rightarrow e \in D) & \Rightarrow \text{opt} d \in D \\
(\forall d_1, d_2 \in D) & \Rightarrow \text{alt} d_1 d_2 \in D \land \text{strict} d_1 d_2 \in D \\
(\forall d_1, d_2 \in D) & \Rightarrow \text{par} d_1 d_2 \in D \\
(\forall d \in D \land n \in \mathbb{N}) & \Rightarrow \text{loop} n d \in D
\end{align*}$$

in which $\mathbb{N}$ denotes the set of natural numbers.

The Coq represents SDs as an inductive type as below, which it enables reasoning by case analysis and induction.

```
Inductive SeqDiag : Set :=
| Skip : SeqDiag
| E : Event -> SeqDiag
| Alt : SeqDiag -> SeqDiag -> SeqDiag
```

2.2 Semantic model

The semantic model of SDs is a set of traces which concludes all the possible execution traces of this diagram. A trace is a sequence of events $(e_1, e_2, \ldots, e_i, \ldots)$. There is at least one trace in semantic model of an SD, although the trace may be an empty trace. The semantics models is defined as follows in Coq:

```
Definition Signal := string.
Definition Lifeline := string.
Definition Message := Signal + Lifeline + Lifeline.
Definition Event := Kind * Message.
Definition Trace := list Event.
Definition Model := set Trace.
```

2.3 Denotational function

Let $D$ represent the set of SDs and $Mo$ represent the semantics models. The semantics of SDs is defined by a function $\text{interp}: D \rightarrow Mo$ which accepts an SD as argument and yields its denotational model.

The semantics of the empty diagram $\text{skip}$ is defined as an empty trace:

$$\models \text{skip} = \{\langle \rangle\} \quad (1)$$

For a diagram which only contains a single event, the semantics is a set with one trace:

$$\models \langle e \rangle = \{\langle e \rangle\} \quad (2)$$

For the choice operator $\text{alt}$, $\text{alt} d_1 d_2$ means executing diagram $d_1$ or $d_2$. Its semantics is the union of $\models [d_1]$ and $\models [d_2]$:

$$\models [\text{alt} d_1 d_2] = \models [d_1] \cup \models [d_2] \quad (3)$$

Since $\text{opt} d = \text{alt} \text{skip} d$, the semantics of operator $\text{opt}$ is

$$\models \text{opt} d = \{\langle \rangle\} \cup \models [d] \quad (4)$$

The $\text{strict}$ operator defines the strict sequence of two diagrams. $\text{strict} d_1 d_2$ means executing $d_1$ before $d_2$. Let notation $\cdot$ denotes the conjunction of two traces that the second trace comes immediately after the first trace. For example, if $t_1 = \langle 1, 2 \rangle$ and $t_2 = \langle 3, 4 \rangle$ then $t_1 \cdot t_2 = \langle 1, 2, 3, 4 \rangle$.

Let $H$ be a trace set. $m_1$, $m_2$ are two sets of traces. A $\text{strict}$ operator is defined as follow:

$$m_1 \prec m_2 = \{h \in H \mid \exists h_1 \in m_1, h_2 \in m_2 : h = h_1 \cdot h_2\} \quad (5)$$
The semantics of the *strict* operator is defined as:

\[
[\text{strict } d_1 d_2] = [d_1] \times [d_2] = \{ h \in H \mid \exists h_1 \in [d_1], h_2 \in [d_2] : h = h_1 \cdot h_2 \}
\]  

(6)

Diagram \text{loop } n \ d is to execute diagram \( d \) for \( n \) times. The semantics of loop is defined by a semantics loop construct \( \mu_i \), in which \( i \) is the number of times the loop should be iterated. \( M \) represents the denotational model.

\[
\mu_0 M = \{ () \}
\]

(7)

\[
\mu_1 M = M
\]

(8)

\[
\mu_i M = M \bowtie \mu_{i-1} M, \text{if } i > 1
\]

(9)

With this \text{loop} construct, \text{loop } n \ d is defined as:

\[
[\text{loop } n \ d] = \mu_n [d]
\]

(10)

Diagram \text{par } d_1 d_2 indicates executing diagram \( d_1 \) and \( d_2 \) concurrently. Let \( x, y \) be events and \( t, r \) are traces. The constructor of \text{par} is defined as:

\[
\varepsilon \parallel t = t \parallel \varepsilon = t
\]

(11)

\[
x \cdot t \parallel y \cdot r = (\{ (x) \} \times (t \parallel y \cdot r)) \cup (\{ (y) \} \times (x \cdot t \parallel r))
\]

(12)

\[
[\text{par } d_1 d_2] = \{ t_1 \parallel t_2 \mid t_1 \in [d_1] \land t_2 \in [d_2] \}
\]

(13)

In Coq, the denotational semantics is presented as a recursive function \text{interp} as below:

Fixpoint interp (D : SeqDiag) {struct D} : Model :=
match D with
| Skip => (⟨nil⟩)::nil
| E e => (e::nil)::nil
| Strict SD1 SD2 => unionModel (interp SD1) (interp SD2)
| Alt SD1 SD2 => set_union Trace_dec (interp SD1) (interp SD2)
| Opt SD => set_union Trace_dec (⟨nil⟩::nil) (interp SD)
| Loop n SD => LoopStrict n (interp SD)
| Par SD1 SD2 => interleavedModel (interp SD1) (interp SD2)
end.

The constructor of \text{unionModel} and \text{set_union} is the implementation of \text{strict} constructor that inner joins all the traces of two trace sets; \text{set_union} is an inner function in the standard library of Coq which is used here to union two trace sets; \text{LoopStrict ?} \ n \ m is to repeat the trace in \( m \) for \( n \) times. For example, let \( a, b, c \in T, m = \{ a, b, c \} \), then the result of computing \text{LoopStrict 2} \ m = \{ a \cdot a, a \cdot b, a \cdot c, b \cdot a, b \cdot b, b \cdot c, c \cdot a, c \cdot b, c \cdot c \}. \text{LoopStrict} is the implementation for the \text{loop} constructor and \text{interleaveModel} is the implementation for the \text{par} constructor.

### 3 Operational semantics

In this section we describe the operational semantics of SDs, including big-step semantics and small-step semantics. Big-step semantics is also called natural semantics. It specifies the entire transition from an initial configuration to a final value. In SDs, the configuration is a pair constructed by an initial trace and the diagram to execute. The final value is the execution traces. Small-step semantics is also called reduction semantics. It specifies the operation one step at a time. There is a set of rules that we continue to apply to configurations until reaching a final configuration [12].

#### 3.1 Big-step semantics

Let \( T \) be the set of traces and \( D \) be the set of SDs. An execute system is a triple

\[
\text{execute}[:_\_ _] \in T \times D \times T
\]

(14)

The \( D \) in (14) represents the diagram to execute and the first \( T \) is the initial trace, the second \( T \) represents the trace after execution. Let \( t \) be the initial trace for all the definition follow. For the empty diagram \text{skip}, the trace will not change after execution.

\[
\text{execute}(t, \text{skip}, t)
\]

(15)

For a single-event diagram \( e \), the result trace equals to the initial trace \( t \) connected with event \( e \) after execution.

\[
\text{execute}(t, e, t, \langle e \rangle)
\]

(16)

The choice operator \text{opt} specifies a diagram that is alternatively \text{skip} or \( d \). There exist two results after execution. For all \( t \in T \),

\[
\text{execute}(t, \text{opt } d, t)
\]

(17)

\[
\begin{align*}
\text{execute}(\langle \rangle, d, t) \\
\text{execute}(t, \text{opt } d, t, t)
\end{align*}
\]

(18)

If it executes as \text{skip}, the result will be the same as initial trace. If it executes \( d \), the result trace will be \( t \) followed by the result trace of diagram \( d \).

The execution of choice operator \text{alt} is similar to \text{opt}. There also exist two rules to execute \text{alt}. For all \( t \in T \),

\[
\begin{align*}
\text{execute}(\langle \rangle, d_1, t) \\
\text{execute}(t, \text{alt } d_1 d_2, t, t)
\end{align*}
\]

(19)

\[
\begin{align*}
\text{execute}(\langle \rangle, d_2, t) \\
\text{execute}(t, \text{alt } d_1 d_2, t, t)
\end{align*}
\]

(20)

The trace after execution may be the initial trace connecting result trace of \( d_1 \) or \( d_2 \).

Diagram \text{strict } d_1 d_2 means executing \( d_2 \) immediately after the execution of \( d_1 \). The result trace is a conjunction
among initial trace \( t_1 \), the result trace of \( d_1 \) and the result trace of \( d_2 \). For all \( t_1, t_2 \in T \),

\[
\frac{\text{execute}(\langle \rangle, d_1, t_1) \land \text{execute}(\langle \rangle, d_2, t_2)}{\text{execute}(t_1, \text{strict} d_1 d_2, t_1, t_1, t_2)}
\]

(21)

Iteration diagram \( \text{loop} n d \) means executing the diagram \( d \) iteratively for \( n \) times. Since big-step semantics can only describe termination semantics, we only consider finite loop \((n \neq \infty)\). We have for all \( t, t' \in T \),

\[
\frac{\text{execute}(\langle \rangle, d, t) \land \text{execute}(\langle \rangle, \text{loop}(n) d, t')}{\text{execute}(t, \text{loop}(n) d, t, t \cdot t')}
\]

(22)

\[
\text{execute}(t, \text{loop} 0 d, t)
\]

(23)

\( S_n \) in (22) means the successor of \( n \) which is \( n + 1 \). For all diagram \( d \), the execution of \( \text{loop}(n) d \) is the same as executing \( \text{loop}(n) d \) strictly after executing simple diagram \( d \). Executing the loop for zero time won’t change the trace.

For operator \( \text{par} \), we have for all \( t_1, t_2 \in T \),

\[
\forall t \in t_1 \parallel t_2 \text{execute}(t, \text{par} d_1 d_2, t_1 \cdot t)
\]

(24)

### 3.2 Small-step semantics

A relation \( \rightarrow \) (called \( \text{red} \) in our system in Coq) is defined to describe the one-step execution of SDs.

\[
(D \times T) \rightarrow (D \times T)
\]

\( \rightarrow \) is a relation between two pairs. The first pair represents the initial diagram and the trace, while the second pair is the diagram and trace after one-step execution.

(25) to (38) are reduction rules for each operator.

(25)

\[
\forall t \in T, (d_1, t_1) \rightarrow (d_1', t_1)
\]

Diagram \( \text{loop} n d \) represents executing diagram \( d \) strictly for \( n \) times. There are two reduction rules

\[
(\text{loop} (S n) d, t) \rightarrow (\text{strict} d (\text{loop} (n) d), t), \text{if } n > 0
\]

(33)

\[
(\text{loop} 0 d, t) \rightarrow (\text{skip}, t)
\]

(34)

Diagram \( \text{loop} n d \) only describes finite loop, since \( n \in \mathbb{N} \) and \( n \neq \infty \). Although small-step semantics can represent non-terminating condition, big-step semantics is always terminating. We only consider the terminated case for all the semantics of operators, so the two kinds of operational semantics are equivalent (see section 4).

For constructor \( \text{par} \), we have

\[
\forall t, t \in T, (d_1, t_1) \rightarrow (d_1', t)
\]

(35)

\[
(\text{par} d_1 d_2, t_1) \rightarrow (\text{par} d_1' d_2, t_2)
\]

(36)

\[
(\text{par} d_1 d_2, t_1) \rightarrow (\text{par} d_1' d_2', t_2)
\]

(37)

\[
(\text{par} d_1 d_2, t_1) \rightarrow (d_1, t_1)
\]

(38)

\[
(\text{par} d_1 d_2, t_1) \rightarrow (d_1, t_1)
\]

(39)

Correspondingly, we introduce the function \( \text{star} \) and predicate \( \text{terminates} \) from [4] in Coq to describe the reflexive transitive closure and termination property.

Codes below shows the definition of \( \rightarrow \) relation, reflexive transitive closure and the prediction of termination in Coq.
There exist two induction hypotheses: H1 : execute(⟨⟩, d′, x1) and H2 : execute(⟨⟩, loop n′ d′, x2).
By applying formula (22) to hypothesis H1 and H2, we get execute(⟨⟩, loop (n+1) d, x1 · x2) which is the desired result.

4.3 Equivalence between the small-step operational semantics and the big-step operational semantics

Section 3 introduces execute function to represent the big-step semantics and → (red) relation which represents the small-step semantics. The soundness and completeness of these two semantics are presented as follows.

Theorem 3 : ∀d ∈ D, ∀t ∈ T,
if execute(⟨⟩, d, t), then (d, ⟨⟩) → (skip, t)
For all diagrams d, all traces generated by the bigstep can also generated by the smallstep.

Theorem 4 : ∀d ∈ D, ∀t ∈ T,
if (d, ⟨⟩) → (skip, t), then execute(⟨⟩, d, t)
For all diagrams d, all traces reasoned by the small-step can also generated by the bigstep.

To prove theorem 3 and theorem 4, induction of diagram d is the first step. In theorem 3, execution trace t can be obtained by applying execution rules to execute(⟨⟩, d, t) repeatedly. Then t is subtitled to (d, ⟨⟩) → (skip, t) and proved by applying red rules. Proof of theorem 4 is similar to that of theorem 3.

4.4 Equivalence between the denotational semantics and the operational semantics

Theorem 5 : ∀d ∈ D, ∀t ∈ T,
if execute(⟨⟩, d, t), then t ∈ [d]
Theorem 5 ensures the soundness of the equivalence between the two semantics which means if t is an execution trace of the given diagram d, t must be in [d]

Theorem 6 : ∀d ∈ D, ∀t ∈ T,
if t ∈ [d], then execute(⟨⟩, d, t)
Theorem 6 proves the completeness of the equivalence between the two semantics which means for all the trace t in [d], we have execute(⟨⟩, d, t).
The set of execution traces can be achieved by executing denotational function interp defined in our system. In the above case, it equals to \([d] = \{\langle \text{id} \cdot \text{rid} \cdot \text{spwd} \cdot \text{rpwd} \cdot \text{slogin} \cdot \text{rlogin} \rangle, \langle \text{id} \cdot \text{rid} \cdot \text{spwd} \cdot \text{rpwd} \cdot \text{slogin} \cdot \text{rlogin} \rangle, \langle \text{id} \cdot \text{rid} \cdot \text{spwd} \cdot \text{rpwd} \cdot \text{slogin} \cdot \text{rlogin} \rangle, \langle \text{id} \cdot \text{rid} \cdot \text{spwd} \cdot \text{rpwd} \cdot \text{slogin} \cdot \text{rlogin} \rangle\}\)

Every trace in denotational semantics can be got by applying execution rules repeatedly, since the correctness of rules has been proved. Detail steps to reason about trace \(\langle \text{id} \cdot \text{rid} \cdot \text{spwd} \cdot \text{rpwd} \cdot \text{slogin} \cdot \text{rlogin} \rangle\) is listed below. For clarity, we assume that

\[d_1 = \text{strict (strict \text{id} \cdot \text{rid})} (\text{strict \text{spwd} \cdot \text{rpwd}}),\]

\[d_2 = \text{strict \text{slogin} \cdot \text{rlogin}}\]

\[d_3 = \text{strict \text{scmd} \cdot \text{rcmd}}\]

The following formula exists by applying (26) and (31)

\[(d_1, \langle \rangle) \searrow (\text{skip}, (\text{id} \cdot \text{rid} \cdot \text{spwd} \cdot \text{rpwd}))\]

\[(d_2, \langle \rangle) \searrow (\text{skip}, (\text{slogin} \cdot \text{rlogin}))\]

\[(d_3, \langle \rangle) \searrow (\text{skip}, (\text{scmd} \cdot \text{rcmd}))\]

Then we can achieve trace

\[\text{strict \text{d1} (\text{alt (strict \text{d2} (\text{opt \text{d3}}))}), (\langle \rangle) \rightarrow (\text{strict \text{skip} (\text{alt (strict \text{d2} (\text{opt \text{d3}}))})}, (\text{id} \cdot \text{rid} \cdot \text{spwd} \cdot \text{rpwd})) \rightarrow (\text{alt (strict \text{d2} (\text{opt \text{d3}})}), (\text{id} \cdot \text{rid} \cdot \text{spwd} \cdot \text{rpwd}) \rightarrow (\text{strict \text{d3} (\text{opt \text{d3}})}), (\text{id} \cdot \text{rid} \cdot \text{spwd} \cdot \text{rpwd} \cdot \text{slogin} \cdot \text{rlogin})\rightarrow (\text{opt \text{d3}}, (\text{id} \cdot \text{rid} \cdot \text{spwd} \cdot \text{rpwd} \cdot \text{slogin} \cdot \text{rlogin})) \rightarrow (\text{d3}, (\text{id} \cdot \text{rid} \cdot \text{spwd} \cdot \text{rpwd} \cdot \text{slogin} \cdot \text{rlogin})) \rightarrow (\text{skip}, (\text{id} \cdot \text{rid} \cdot \text{spwd} \cdot \text{rpwd} \cdot \text{slogin} \cdot \text{rlogin})) \rightarrow (\text{scmd} \cdot \text{rcmd}))\]

Similarly, all traces in denotational semantics can be achieved by applying these execution rules and they can be proved mechanically.

6 Related work

There are commonly two ways to define formal semantics of SDs. One is to transmit to an existing formalism. For example, [5] encoded SD into the Prototype Verification System (PVS). [6],[7] introduced a formal semantics for SD by means of Petri nets as a formal model in order to express the partially ordered and concurrent behaviour of the diagrams. [1],[2],[18] formalized SD into state machine, and [17] proposed a novel formalization method of SD based on the temporal description logics(TDLs) since it can describe both static and dynamic domain knowledge. But none of above proved the correctness(semantic preservation) of their formal semantics.

Another way is to define a new formal semantics models. [15] summarized the proposed semantics of SDs in last 8 years. [3] proposed an operational semantics, where an interaction automaton was produced by unwinding the Interaction. It captured not only standard composi-
tion operators but also the negation and assertion operators. But all the semantics was described on paper, they were not defined mechanically.

[20],[21] adapted template semantics to describe the operators and combined fragments of SD, which were new structured control constructs of UML 2. However, the correctness of the operational semantics had never been discussed.

[14] defined a part of mechanized semantics for SDs, and the operational semantics was implemented in the Maude language. However, only parts of proofs about the equivalence between the denotational semantics and the operational semantics were written on the paper. We are unaware of anyone representing mechanized semantics of both the denotational semantics and the operational semantics, and mechanically proved the related properties.

7 Conclusion and future work

In this paper, mechanized semantics of a subset of SDs is defined in a proof assistant-Coq. The syntax, denotational semantics of operators, big-step semantics, and small-step semantics are defined. The desired properties of SDs are also proved, including the termination of diagram. The correctness of operational semantics compared to denotational semantics is verified mechanically.

Several extensions can be considered. One of our future work is to add negative operators into consideration, such as neg. Both denotational semantic model and operational semantics need to be expand. Another piece of the future work is to optimize the loop operator in order to describe infinite loop and prove the properties about divergence of red.

Acknowledgement

This work is supported by National Natural Science Foundation of China (No.61070226 and No.61003181)

References


