Sematics of UML Sequence Diagrams in PVS

Demissie B. Aredo

Department of Informatics, University of Oslo
Institute for Energy Technology
P.O.Box 173, N-1751 Halden, Norway
demissie@hrp.no

Abstract. In this paper, we present formal semantics of UML (Unified Modeling Language) *sequence diagrams* using the PVS (Prototype Verification System) [8] as an underlying semantic foundation. We give a formal definition of a trace-based semantics [5] of UML sequence diagrams; i.e. a sequence diagram is interpreted as a set of traces of events that may occur in the realization of the interaction specified by the sequence diagram. This work is a part of a long-term vision to explore how the PVS tool set could be used to underpin practical tools for analysis of models in UML. It also contributes to the ongoing effort to provide formal semantics of UML, with the aim of clarifying and disambiguating the language as well as supporting the development of semantically based tools.

Keywords : formal semantics, UML, PVS, sequence diagram

1 Introduction

The Unified Modeling Language (UML) [12] is a collection of object-oriented modeling notations that has been standardized by the OMG (Object Management Group) [7]. It is a notation for specifying, visualizing and documenting artifacts of software-intensive systems, and has rapidly become an important industrial standard. The lack of formal semantics of UML notations renders its limitations in the context of rigorous formal reasoning about UML models.

There are numerous attempts at giving a formal semantics to fragments of the UML [3, 4, 14]. A distinguishing feature of our work is the goal of wishing to utilize existing, powerful tools to analyze UML models and instances of those models. Specifically, we have chosen to focus on the PVS environment [8] for two main reasons: Firstly, it supports semantics notions that are important for specifying reactive systems. For instance, the PVS specification language (PVS-SL) supports notions of sequences, lists, records, etc. that are useful in defining a trace-based semantics of UML sequence diagrams. Secondly, the PVS environment has a powerful tool set, including a *type-checker, theorem-prover*, and *model-checker*, and exploits the synergy between the highly expressive specification language and a theorem-prover that is based on powerful decision procedures.

The rest of this paper is structured as follows: In Section **??**, we give a brief overview over a subset of the PVS environment, the specification language (PVS-SL) and the theorem prover, and demonstrate how they can be used in combination. In Section 3, we discuss main concepts of UML *sequence diagrams* and define their semantics. In Section 4, we define a formal semantics of UML sequence diagram. Finally, in Section 5, we conclude and discuss future research issues.

2 The Prototype Verification System

The Prototype Verification System (PVS) consists of a specification language (PVS-SL)[9], a type checker and a theorem-prover [10] based on a classical, simply typed higher-order logic. It exploits the synergy between a highly expressive specification language and an interactive theorem-prover based on powerful decision procedures. It has a rich type system which has been augmented by *predicate subtype* and *dependent types*: features that render type-checking undecidable; the type-checker may require users to show that their specifications are consistent by generating proof obligations known as *type correctness condition* (TCC's). Specifications in PVS-SL are organized into a set of, possibly parameterized, *theories*. A theory may specify *types, variables and constants, definitions, axioms, theorems* and *assumptions* on its parameters. The PVS environment includes an extensive set of built-in theories, called *preludes*, that provide several useful types, definitions, lemmas etc. A theory may import or 'instantiate' other built-in or user-defined theories.

The *records, functions*, and *sets* type constructors are used extensively in the sequel. A record type is a finite list of fields of a general form `R : TYPE = [#` $a_1 : T_1, \ldots, a_n : T_n$ `#]` where a_i's are accessor functions and the T_i's are types. Given a record `r:R`, function application-like terms $a_i(r)$ or `r`a_i, rather than the conventional 'dot' notation, is used to access the i^{th} field of a record `r`. Record types are similar to tuple types, except that the order of the fields is unimportant. Function types are declared as `F : TYPE =` $[T_1, \ldots, T_n \rightarrow T]$ where T_i's and `T` are type expressions. There are two forms of specifying a set: `pred[T]` and `setof[T]`. Both of them are shorthand for $[T \rightarrow \texttt{bool}]$, and are provided in the prelude. Semantically, they are the same since sets are represented as predicates in PVS.

3 Basic Concepts of UML Sequence Diagrams

The UML notation is comprised of two main categories of modeling elements: static structural elements such as *classes, interfaces*, and *relationships*; and dynamic behavioral elements such as *objects, messages*, *finite state machines*, and *message sequence charts* (or sequence diagrams as they are called in the UML). In the sequel, we exclusively investigate semantics *UML sequence diagrams*. A sequence diagram describes a specific interaction in terms of the set of participating objects and a sequence of messages they exchange as they unfold over

time to effect the desired operation or result. It is useful especially for specifying systems with time-dependent functions such as real-time applications, and for modeling complex scenarios where time dependency plays an important role.

A sequence diagram specifies only a fragment of system behaviour. To specify the complete behaviour of a system, a set of sequence diagrams should be used to specify all possible interactions during its life cycle [1].

Before we formally define semantics of sequence diagrams, we specify some basic notions of sequence diagram, namely, *action, event, message, object*, and *operation*.

3.1 Events

An event is a specification of a significant occurrence that has a location in time and space. In a description of communications among system components, basically we identify three types of events: a *local* operation call, a message *send*, and *receive* event. The send and receive events are instances of remote operation call on the target object.

An event is represented as a record type whose fields consist of: the event identifier which is normally identical to the identifier of the associated message, the identifiers of the sender and the set of receiver of the associated message, a tag that specifies the type of event, the action that will take place, and arguments. Symbolically, event is specified as follows:

```
Events : THEORY
  BEGIN
    Action : TYPE
    EventType : TYPE = {send, receive, local}
    EventID, ObjectID, Parameter : TYPE
    Event : TYPE = [# eventID : EventID,
                      sender : ObjectID,
                      receiver : setof[ObjectID],
                      eventType : EventType,
                      action : Action #]
  SendEvent : TYPE = {e:Event | eventType(e) = send}
  ReceiveEvent : TYPE = {e:Event | eventType(e) = receive}
  LocalEvent : TYPE  = {e:Event | eventType(e) = local}
  causal(es:SendEvent, er:ReceiveEvent) : bool =
           (es = er WITH ['eventType := eventType(es)])
  END Events
```

The WITH construct is similar to function overriding in Z [15]. The `causal()` operation checks whether a given pair of `SendEvent` and `ReceiveEvent` events constitutes a valid message.

3.2 Messages

A message is a specification of a communication among objects, or an object and the system and its environment, that conveys information with the expectation

that activity will ensue. It specifies the roles of the sender and receiver objects, as well as the associated action which specifies the statement that causes the communication to take place.

A message can be either a signal (asynchronous) or an operation call (synchronous). Since sending a signal and calling an operation are similar at logical level, we consider synchronous communications. In our framework, sending and receiving of a message are considered as two distinct instances of events. An event involves exactly two (not necessarily distinct) objects. In case of iterative message passing and message broadcast, each communication is modeled separately. Hence, we model a message as a pair of message sending event and message receiving event. The correspondence between the send and receive events has to be established uniquely. The operation to invoked (the action), and its parameters are extracted from the events.

An important constraint on message communication is the causality requirement which is formalized as a relation between message send events and the corresponding message receive events - a requirement that must hold for every trace satisfying the causality predicate defined in `Objects` theory

The UML notation supports modeling of the real-time concept. To specify behaviors that involve the notion of time, we need to adorn the event record with a field to store the time of event occurrence (these corresponds to the *sendTime* and *receiveTime* in UML meta-model UML v1.3 pp. 3-98). In the sequel, however, we consider only the time sequence of event occurrences, and there is no notion of global clock.

3.3 Objects

An object participating in a given interaction exhibits observable properties of its class(es). In a sequence diagram, existence of an Object is depicted by an object box and its *'life-line'*. A life-line is a vertical line that shows the existence of an object over a given period of time, object creation and/or destruction during the interaction specified by the sequence diagram, and the ordering of events that may occur on the object. But, it does not specify the exact time elapsed between occurrences of two events. To model the time notion of communication, we stamp every event by the time of its occurrence. For a message `m`, time information like `m.sendTime`, and `m.receiveTime` is modeled by the time of occurrences of the related message send and message receive events respectively. This time information is used to express timing constraint on traces of events, e.g. minimum time between occurrences events.

An object specifies an entity on which an operation can be invoked and which has a state that stores the effects of operations. An object may have a set of attribute values, and is connected to a set of links, where both sets conform with the specifications of its classes, and implement the current state of the object. We define the semantics of an object as a trace of events (operation invocations) satisfying the causality condition, and represent an object as a record type whose fields include: a unique object identifier, a non-empty set of classes, a set of

attribute links, a set of operations, and a set of traces of events that may occur on the object.

```
Objects : THEORY
  BEGIN
    IMPORTING Events, Classes
    ObjectID, Operation, AttributeLink : TYPE
    Trace : TYPE = list[Event]
    Object_Status : TYPE = {new, destroyed, transient}
    ObjectRec : TYPE = [# objectID : ObjectID,
    classes : finite_set[Class],
    slots : finite_set[AttributeLink],
    operations : finite_set[Operation],
    trace : set[Trace] #]
    class_exist?(obj:ObjectRec) : bool =
                FORALL obj: NOT empty?(classes(obj))
    status : [Object → Status]
    all_operations(obj:Object) : setof[Operation] =
           {opr : Operation | ∃ (c:Class): obj(classes)(c) }
    causal?(t:Trace): bool = FORALL (er:ReceiveEvent): member(er, t)
              ⇒ (EXISTS (es:sendEvent) : member(es, upto(er, t)) &
              match(es,er))
    Object: TYPE = {obj : ObjectRec | causal?(obj(trace)) &
                                      class_exist?(obj) &
                                      all_operations(obj) }
    one_class : AXIOM (∀(obj : Object) : classes(obj) ≠ ∅)
  End Objects
```

In a paradigm where multiple and dynamic classification is allowed, an object may gain or lose a class during execution. However, an object must be connected to at least one 'direct' class which declares its structure and behaviour. The axiom `one_class` states that for every object, at any point in time, there must exist at least one class of which the object is a direct instance. Other properties such as the conformance of the set of link ends of an object to the set of association ends of its class(es) can similarly be stated and proved correct. A status of an object at the end of the interaction (`new, destroyed` or `transient`), and the set of operations that can be invoked on the object, cab be extracted from the object record type.

3.4 Traces of Events

A trace is a sequence of events that satisfies some predicates on events and state variables such as the causality predicate. In the sequel, we consider prefix-closed finite traces. Theoretically, traces can be of finite or infinite lengths. In practice, however, finite trace semantics suffice to model behaviour of a system over a finite time interval (assuming that iterations in sequence diagrams are finite).

In PVS-SL, a predefined parameterized theory specifies the `list` abstract data type which is synthesized into PVS type theory that models the standard

`list` type along with its operations. In the `Traces` theory, we specify a trace as a prefix-closed list of events. We also define some auxiliary operations on lists, and state basic properties of traces, and sequence diagrams they model as predicates. The `prefix()` and `prefix_upto_n()` functions, for example, specify the correspondence between send and receive events that comprise a message. Moreover, if a trace is projected on a given set of events that occur on an object, the causality condition will be maintained.

```
Traces : THEORY
  BEGIN
  IMPORTING Events
    tr, tr1, tr2 : VAR list[Event]
    x, ev, e1, e2 : VAR Event
    prefix(tr1, tr2) : bool = ∃ tr : (tr2 = append(tr1,tr))
    prefix_upto_n(n:nat, tr:Trace) : RECURSIVE list[T] =
                CASES tr OF
                null : null,
                cons (x, tr1) :
                     IF n = 0  THEN null
                     ELSE cons(x, prefix_upto_n(n-1, tr1))
                     ENDIF
                ENDCASES
  MEASURE length(tr)
  causal?(tr:Trace): bool =
                   ∀ (er:ReceiveEvent): member(er,tr) ⇒
                     (∃ (es:sendEvent) :
                        member(es, prefix_upto_n(rank(er), tr)) &
                        match(es,er))
   precedes(e1, e2, tr) : bool =
           prefix(prefix_upto_n(rank(e1),tr), prefix_upto_n(rank(e2),tr))
   projection : [list[T], setof[T] → list[T]] = filter
   Trace : TYPE = {tr : list[Event] | causal?(tr)}
  END Traces
```

4 Semantics of Sequence Diagrams

Once the basic concepts of sequence diagrams are formally specified, semantics of the sequence diagram is modeled by a PVS theory that composes the the constituents. Properties of the system specified by the sequence diagram should (not) exhibit, e.g. system invariant and constraint, as *predicates, axioms* and *conjectures*. This approach is in line with the specification style of PVS, where an entity should be defined before it can be used, as there is no forward reference in PVS. A sequence diagram is represented by a record type whose fields are consisting of

- the identifier of a sequence diagram
- a set of objects participating in the interaction specified by the sequence diagram and conforming to the ClassifierRoles specified by its context

- a set of traces of events that models the interaction

A sequence diagram is modeled as a predicate subtype of the record type.

```
SeqDiagrams : THEORY
BEGIN
   IMPORTING Objects, Traces
   SeqDiagramID: TYPE
   SeqDiagRecord : TYPE = [# seqDiagID : SeqDiagID,
                            objects : setof[Object],
                            traces : setof[Trace] #]
   sqr : VAR SeqDiagRecord
   ev : VAR Event
   tr : VAR Trace
   obj : VAR Object
   objset : VAR setof[Object]

   proj?(sqr,obj,tr): bool = traces(sqr)(tr) & objects(sqr)(obj) ⇒
                projection(tr,list2set(trace(obj))) = trace(obj))

   comp?(sqr, tr, ev): bool = (traces(sqr)(tr) & member(ev, tr)) ⇒
        (EXISTS obj: (objects(sqr)(obj) &
                member(operation(action(ev)), all_operations(obj))))

   prefix_closed?(trset : setof[Trace]) : bool =
                member(null, trset) &
                (member(cons(ev,tr),trset) ⇒ member(tr, trset)

   SeqDiagram : TYPE = {sqr : SeqDiagRecord |
                prefix_closed?(traces(sqr)) &
                every(causal?)(traces(sqr))&
                every(proj?)(sqr, objects(sqr), traces(sqr)) }

thm: THEOREM (member(tr,traces(sqr)) & subset?(objset,objects(sqr))
   ⇒ every(proj?)(sqr,tr,objset) & every(comp?)(sqr,tr,ev))}
END SeqDiagrams
```

The `list2set` is a predefined function that converts a list into a set. A set of traces of events is possible instances of runs of the system specified by the sequence diagram if and only if it is a prefix-closed and satisfies the causality requirement `causal?`. Moreover, for every allowed trace and every object participating in the interaction specified by the sequence diagram, the projection of the trace to the set of events of the object must be a allowable trace of the object, `proj?`. The composition predicate `comp?` specifies that for every event in a trace, there must exist an object on which the operation associated with the event is invoked. Other constraints that specify, well formedness rules and the relationships between elements of sequence diagram are stated similarly.

5 Conclusion and Future Work

In this paper, we present a work done on formalization of UML sequence diagrams by using the higher-order logic of PVS-SL. The choice of PVS is dictated by its suitability to handling fundamental OO modeling concepts such as polymorphism, and inheritance. This work contributes to the ongoing effort to provide formal semantics of UML notations with the aim of clarifying and disambiguating the language as well as supporting the development of semantically based tools. It is also a part of a long-term vision to explore how the PVS tool set could be used to underpin practical tools for analysis of UML models.

A purpose of formalization of a modeling language is not only to make specifications precise, but also to provide automated support for rigorous model analysis. To provide an automated support, a prototype of a multi-formalism platform, called the *Integrator* [16] is developed. The platform integrates the UML tool - Rational Rose [13], and the PVS toolkit to support the functionality necessary to cover the whole development cycle of open distributed systems from requirement capture to final code production. PVS-SL is used in the platform as semantics foundation and not as specification language, and hence the user need not have an in-depth knowledge about the PVS formal notation and proof system.

There are several research works on the formalization of UML notations [3], [4], [14], using formalism such as *Z* [15] as the underlying semantic notation. But Z has limitations to deal with the dynamic aspects of systems, and the tool support for *Z* is weak. A similar work was done by Duterrte [2] on encoding of CSP [5] in PVS. A distinguishing feature of our work is the integration of existing tools, the PVS and UML tools, in order to support developers to design and analyze UML models.

In the future, we will introduce refinement proof rules, and validation rules necessary for rigorous model analysis in the context of distributed systems, and extend our work to other dynamic constructs of the UML such as statecharts.

Acknowledgements
I am grateful to Stuart Kent, Olaf Owe, Ketil Stølen, and Wenhui Zhang for reading earlier drafts of this paper and for their invaluable comments. This work is financed by the Research Council of Norway in the framework of the ADAPT-FT project.

References

1. R. Breu, R. Grosu, C. Hofmann, F. Huber, I. Kruger, B. Rumpe, M. Schmidt, W. Schwerin, *Exemplary and Complete Object Interaction Descriptions*, in Proceedings of OOPSLA'97 Workshop on Object-oriented Behavioral Semantics, Atlanta, Georgia, October 1997.
2. B. Dutertre, S. Schneider, *Embedding CSP in PVS: An Application to Authentication Protocols*, Technical Rep. 736, Deprtment of Computer Science, Queen Mary and Wesfield College, University of London, May 7, 1997.

3. A. Evans, *Reasoning with UML class diagrams*, In WIFT'98, IEEE Press 1998.
4. R. B. France, A. Evans, K. Lano, and B. Rumpe *The UML as a Formal Modeling Notation*, Computer Standards & Interfaces, 19 (1998), p. 325-334.
5. C. A. R. Hoare, *Communicating Sequential Processes*, Prentice Hall, 1985.
6. O. Owe, I. Ryl, *The Oslo University Notation: A Formalism for Open, Object-Oriented, Distributed Systems*, Reserach Report No. 270, Department of Informatics, University of Oslo, Norway, August 1999.
7. Object Management Group Inc. *OMG Unified Modeling Language Specification*, version 1.3, June 1999, available at http://uml.shl.com/artifacts.htm.
8. S. Owre, N. Shankar, J. Rushby, D. W. Stringer-Calvert*PVS Language Reference*, http://pvs.csl.sri.com/manuals.html, Version 2.3, September 1999.
9. S. Owre, N. shankar, J. M. Rushby, *The PVS Specification Language*, Computer Science Lab., SRI International, April 1993.
10. N. shankar, S. Owre, J. M. Rushby, *The PVS Proof-checker: A reference Manual*, Computer Science Lab., SRI International, April 1993.
11. J. Crow, S. Owre, J. Rushby, N. Shankar, M. Srivas,*A Tutorial Introduction to PVS*, WIFT'95: Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida, April 1995.
12. The UML Group, *The Umified Modeling Language, Version 1.1*, Rational Software Corporation, Santa Clara, CA-95051, USA, July 1997.
13. http://www.rational.com/products/rose/
14. M. Shroff, and R. B. France, *Towards a formalization of UML Class Structures in Z*, in the Proceedings of the COMPSAC'97, 1997.
15. J.M. Spivey, *The Z Notation: A Reference Manual*, Prentice-Hall International, 1989.
16. I. Traoré, *The UML Specification of the Integrator*, Technical Report RS-275-IFI, Department of Computer Science, University of Oslo, August 1999.