Journal of Universal Computer Science, vol. 8, no. 7 (2002), 674-697
submitted: 16/1/02, accepted: 22/7/02, appeared: 28/7/02 © J.UCS

A Framework for Semantics of UML Sequence Diagrams in PVS

Demissie B. Aredo
Department of Informatics, University of Oslo
P. O. Box 1080 Blindern, N-0316 Oslo, Norway
demissieifi.uio.no

Abstract: This paper presents a framework for representing formal semantics of a subset of the Unified Modeling Language (UML) notation in a higher-order logic, more specifically semantics of UML *sequence diagrams* is encoded into the Prototype Verification System (PVS). The primary objective of our work is to make UML models amenable to rigorous analysis by providing their precise semantics. This approach paves a way for formal development of systems through a systematic transformation of UML models. This work is a part of a long-term vision to explore how the PVS tool set can be used to underpin practical tools for analyzing UML models. It contributes to the ongoing effort to provide mathematical foundation to UML notations, with the aim of clarifying the semantics of the language as well as supporting the development of semantically-based tools.

Key Words: Formal Semantics, UML, PVS, Formal Methods, Object-Orientation
Category: D.3.1, D.1.5, D.2.4

1 Introduction

The Unified Modeling Language (UML) [Rumbaugh et al., 1999, OMG, 1999, Booch et al., 1999] is an object-oriented modeling language that consists of a comprehensive set of notations. It is an industry standard modeling language (standardized by the Object Management Group (OMG)) for specifying, visualizing, and documenting artifacts of software intensive systems. Among the distinguishing properties of UML is its capacity to unify a collection of notations for object-oriented modeling - a property that may raise several fundamental issues in the context of software engineering.

Compared to other object-oriented modeling languages in software engineering, UML is more precisely defined and contains a great deal of formal specification notations, for instance, the use of the Object Constraint Language (OCL) [Warmer and Kleppe, 1999] for constraint specification. However, it is not formal *enough* to address problems that relate to the lack of precision [Evans et al., 1998] and suffers from the major drawbacks of object-oriented methodologies - their limitation in the context of formal reasoning. The semantics of UML constructs is expressed in meta-models (descriptions of UML in UML) and natural language. Although the meta-models capture a precise notion of the abstract syntax of the UML modeling elements, they do little in addressing problems related to interpretation of non-trivial UML constructs [Evans et al., 1998].

The lack of formal semantic models for graphical UML constructs renders limitations in the context of rigorous model analysis and in developing semantics-based CASE tools [Whittle, 2000, Evans et al., 1998]. Consistency checks provided by currently available CASE tools are, for instance, limited to very simple syntactic checks, such as consistency of naming across models. Great improvements would have been achieved had tools been augmented with deeper semantic definitions for UML models [Whittle, 2000]. Formal methods provide the rigor that is lacking in graphical UML notations. Providing formal semantic models to constructs of a modeling language enables us to identify and remove ambiguities, deficiencies, and inconsistencies from the language. Defining formal semantics for modeling constructs of a graphical language like UML is also a prerequisite for developing semantically based tool support.

In the sequel, we propose semantics definition for UML sequence diagrams in the PVS specification language (PVS-SL) [Owre et al., 1993, Owre et al., 1999a]. We describe a general framework for formalization of UML diagrams, and an approach that involves graphical notations and formal methods to facilitate rigorous model analysis. The approach can readily be used to support system validation and verification. Our reference is the currently available standard documentation for the Object Management Group UML [OMG, 1999]; the informal semantics and the collection of well-formedness rules provided in the documentation. The PVS environment is chosen as an underlying semantic foundation for the following main reasons. Firstly, PVS provides general semantic notions necessary to model reactive systems. For instance, it supports the notions of *sequences, lists, records*, etc. that are crucial for providing trace-based semantic models for UML sequence diagrams. Secondly, the PVS environment has a powerful tool set consisting of a *type-checker*, a *theorem-prover*, and *model-checker*.

Usually, a model given in a single sequence diagram results in only a partial specification, i.e. only subsets of the set of attributes and operations can be derived from a given sequence diagram. To provide a specification of a wide range of interactions in a system, several sequence diagrams should be used in combination. Composition of message sequence diagrams is dealt with in the literature, e.g. see works of Haugen [Haugen, 1997], and Gunter *et al* [Gunter et al., 2001]. Moreover, to obtain a detailed and more complete description of both structural and behavioural aspects of a system, it is necessary to combine several modeling techniques such as class diagrams, statecharts, and sequence diagrams. A class diagram provides structural description of classes and relationships among their objects; a statechart diagram describes dynamic behavior of a component; and a sequence diagram specifies interactions among the components. The UML notation is a combination of these modeling techniques and emphasizes their integrated use to capture properties of systems from different viewpoints. The works of Reggio *et al* [Reggio et al., 2000], Blair *et al* [Blair and Blair, 1999], and Kammüller *et al* [Kammüller and Helke, 2000] address how different modeling techniques can be used.

The rest of this paper is organized as follows. In Section 2, we briefly review the PVS environment, with emphasis put on the PVS specification language and theorem-prover, and discuss how they can be used together. In Section 3, we propose semantic models for basic concepts of UML sequence diagrams such as *actions, events, messages*, and *objects*. In Section 4, we describe the methodology used in our formalization framework, which includes a bottom-up construction of semantics of UML sequence diagrams. In Section 5, we demonstrate, by an example, the application of our formalization framework to model analysis. Finally, in Section 6, we conclude and discuss future research issues.

2 The PVS Environment

The Prototype Verification System (PVS) [Owre et al, 1999b, Crow et al., 1995] is a formalism for design and analysis of system specifications. PVS consists of a highly expressive specification language, a powerful interactive theorem-prover, a type-checker, and other tools. A particular strength of PVS is its capacity to exploit the synergy between its tools, e.g. the type-checker and the theorem-prover complement each other.

The PVS specification language is based on a classical typed higher-order logic. Its type system contains basic types such as *boolean, nat, integer, real*, etc. and type constructors such as *set, tuple, record*, and *function*. Record, set, and function type constructors are extensively used in the sequel to encode abstract syntactic and semantic domains of UML constructs in PVS. A record constructor 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 T_i's are type expression. For a record `r` of type `R`, i.e. `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 `r`. The structure of *tuple* type is similar to that of record type except that the order of fields is significant in tuples.

A function constructor is of a general form `F : TYPE = [`$D_1, D_2, \ldots, D_n \rightarrow R$`]` where D_i's and R are type expressions, `F` is the set of all functions with domain $D = D_1 \times D_2 \times \cdots \times D_n$ and range R. The set of elements of type `T` is denoted by either `pred[T]` or `setof[T]`, where each of them is a shorthand for `S : [T` $\rightarrow$ `bool]`. As a result, given a set `s:S` and an element `t:T`, membership of `t` in `s` is by the truth value of the expression `s(t)`.

The PVS type system has been augmented by *predicate subtyping* and *dependent typing* mechanisms and supports a richer type system than the standard classical higher-order logic and relies on an original approach to type checking [Dutertre and Schneider, 1997]. Given a type `T` and a predicate `p:[T → Bool]`, a predicate subtype T' = {`t:T | p(t)`} of `T` can alternatively be denoted by `(p)`. Subtyping mechanism complicates type-checking, and yet allows a stronger checks for consistency and invariant in a uniform manner [Crow et al., 995]. Accommodating *partial* functions in the logic of *total* functions, for instance, improves expressive power of the specification language. Subtyping mechanism, however, renders type checking undecidable; as a result of which the type-checker generates proof obligations called *Type Correctness Conditions* (TCC) that requires users to discharge them. Though a great deal of TCCs can be discharged automatically, the more involved ones require interactive use of the theorem-prover.

Specifications in PVS are organized into hierarchies of *theories*. A theory may consist of specification of types, variables, constants, definitions, axioms, and conjectures. PVS supports modularity and reuse by means of parameterized theories that make it possible to specify generic modeling elements. The PVS-SL includes an extensive library of built-in theories, called *preludes*, which provide several useful definitions and lemmas. PVS also allows definition of *Abstract Data Types* (ADTs), from which a complete PVS theory is automatically synthesized during type checking.

The following ADT, for example, specifies the standard *stack* data structure along with its constructors `empty` and `push`, two accessor functions `top` and `pop`, and two recognizers `empty?` and `nonemptystack?` that characterize empty and non-empty stacks respectively.

```
stack[T : TYPE] : DATATYPE
  BEGIN
    empty : empty?
    push (top: T, pop: stack) : nonemptystack?
  END stack
```

From such an ADT, a theory called `stack_adt[T:TYPE]` that consists of axioms, theorems, definitions, etc. is automatically synthesized during type checking and completely specifies the `stack` data type axiomatically. For instance, the following is one of the axioms generated during type checking, and states an invariant property of stacks, i.e. for any stack a *push* operation followed by a *pop* operation leaves the stack unchanged. Symbolically,

```
pop_push_ax : AXIOM (FORALL (x: T, s: stack): pop(push(x,s)) = s)
```

Another invariant property of stacks is that application of two *push* operations followed by two *pop* operations to a given stack leave the stack unchanged. Symbolically,

```
pop_push_th : THEOREM (∀ (x, y: T, s: stack):
                       pop(pop(push(x, push(y, s)))) = s)
```

This theorem can be discharged interactively by invoking the PVS theorem prover. While it is beyond the scope of this paper to explain details of the PVS environment, we have only highlighted some of its key features. For a more detailed discussion of the PVS environment, interested reader should refer to [Crow et al., 1995, Owre et al., 1999a, Owre et al., 1999b]

3 Basic Concepts of UML Sequence Diagrams

The UML sequence diagram is a variant of the classical message sequence charts (MSC) [ITU-TS, 1996]. Sequence diagrams are efficient constructs in modeling dynamic aspects of systems by building up storyboards of scenarios, involving the interacting objects and the messages that may be communicated among them. They show sequences of message passing as they unfold over time, and control flow throughout the interaction to effect a desired operation or result.

A sequence diagram is especially useful to specify reactive systems with time-dependent functions such as real-time applications, and to model complex scenarios where time dependency plays an important role. It is particularly useful technique to visualize dynamic behavior in the context of *use case* scenarios. To

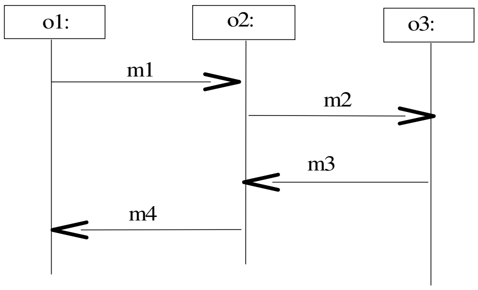


Figure 1: A UML Sequence Diagram

motivate the need for a formal semantics for UML sequence diagrams, let us consider the UML sequence diagram shown in Figure 1. It specifies an interaction among objects `o1, o2`, and `o3`. It constrains messages `<m1, m2, m3, m4>` to occur in that order. The diagram does not, however, state whether any of the messages *must* occur or *may* occur. The sequence `<m1, m2, m4>` is also a valid instance of the interaction modelled by the sequence diagram. In the classical message sequence charts [ITU-TS, 1996], Damm et al [Damm and Harel, 1999] addressed this deficiency by introducing the concept of *temperature* - messages that must occur have *hot* temperature whereas messages that may occur have *cold* temperature. To model dependencies among messages one needs formal representation of sequence diagrams. Suppose that, in Figure 1, message `m4` occurs only if messages `m2` and `m3` occur in that order. This behavior cannot be specified by the graphical notations and induces a strong need for formal semantics.

A sequence diagram specifies only a fragment of system behavior, usually an interaction between objects. To specify the complete behavior of an object or the system as a whole, several sequence diagrams should be used to specify all possible interactions during its life cycle [Breu et al., 1997].

The simplicity of sequence diagrams makes them suitable for expressing requirements as they can easily be understood by the customers, requirement engineers and software developers alike [Whittle, 2000]. The lack of formal semantics for sequence diagrams, however, makes them ambiguous and difficult to interpret. The non-deterministic nature of sequence diagrams also aggravates the ambiguities in their interpretation. The sequence diagram shown in Figure 1, for example, turns to be non-deterministic if message `m2` is removed - the sending of `m1` and `m3` can not be ordered uniquely. As a result, both `<m1.out, m1.in, m3.out, m3.in, m4.out, m4.in>` and `<m3.out, m1.out, m1.in, m3.in, m4.out, m4.in>` are allowable execution traces, where `m.out` and `m.in` denote, respectively, message sending and receiving events for message `m`.

Before we define semantics of sequence diagrams, we need to provide semantic models for the basic concepts, such as *actions, operations, events, messages*, and *objects*.

3.1 Actions and Operations

An action is an invocation of an executable statement that forms an abstraction of a computational procedure that results in a change in the state of the model [OMG, 1999]. It can be realized by sending a message to an object or by modifying a value of an attribute. We represent an action as a record type with the following fields:

- the identifier of the action, normally the name of the associated message
- a list of arguments that determine parameters needed to perform the action
- a set of identifiers of the target objects. This enables us to capture the notion of multi-casting that is used in UML to implement message broadcast.
- a boolean variable that will be used to check whether the action is synchronous or asynchronous.

```
ActionID, ObjectID, ParameterID : TYPE
Action : TYPE = [# actionID : ActionID,
                   args     : finseq[ParameterID],
                   targets  : setof[ObjectID],
                   isAsynch : bool #]
```

where `finseq[]` and `setof[]` are, respectively, types of finite sequences and set of elements of the type given as parameter predefined in PVS library. Note that the PVS specification language is case sensitive, except for built-in identifiers, and hence `actionID : ActionID` is a valid field declaration.

In UML, there are several kinds of actions, namely the *create, destroy, call, return, send, terminate, assignment*, and *uninterpreted* actions. In the UML meta-model, these kinds of actions are specified as subclasses (or specializations) of the generic `Action` class. A `CallAction`, for instance, extends the general structure of `Action` by an attribute which specifies the operation to be invoked, whereas the *CreateAction* specifies the class of which an object is to be created when the action ensue.

To encode classes related by *generalization* relationship into PVS expressions, we use a general scheme that is described next. Consider the class diagram shown in Figure 2(a). `B` is a subclass of `A`. First, the superclass `A` is represented as a PVS record type whose fields consist of the class identifier, a set of attributes, and a set of operations. Then, `B` is encoded in a similar way with one additional field of type `A` that captures inherited parts of `B`, along with its local attributes and operations. The class identifier field of a specialization class is the inherited identifier of the general class. The PVS expressions shown in 2(b) is obtained from the UML class diagram shown in 2(a). The field `asA` (one for every superclass in general case), in the representation of the subclass `B` captures the structure and behavior inherited from the superclass `A`. Detailed discussion of issues related to formal representation of structural UML modeling elements is out of the scope of this paper. Interested readers may refer to relevant works in the literature [Aredo et al., 1999, France et al., 1997, France et al., 1998].

Let's begin by defining structural properties of *operations*, and *call actions*, i.e. remote operation invocation, and requirements on their well-formedness.

```
OperationID, ClassID: TYPE
Operation : TYPE = [# operationID : OperationID,
                      isQuery : bool,
                      parameters : finseq[ParameterID] #]

CallAction: TYPE = [# asAction: Action, operation : Operation #]
```

```
CreateAction: TYPE = [# asAction: Action, class: ClassID #]

param(ca : CallAction) : bool =
              (args(asAction(ca)) = parameters(operation(ca)))
```

The well-formedness rules for UML constructs are stated as predicates. For instance, the predicate `param()` specifies a well-formedness requirement on call actions, i.e. for any call action, the number and type of its arguments must match the parameters of the associated operation. Strictly speaking, call actions are instances of `CallAction` that fulfill all requirements, including well-formedness rules. That is, the set of elements for which all the associated predicates holds - a predicate subtype of `CallAction`.

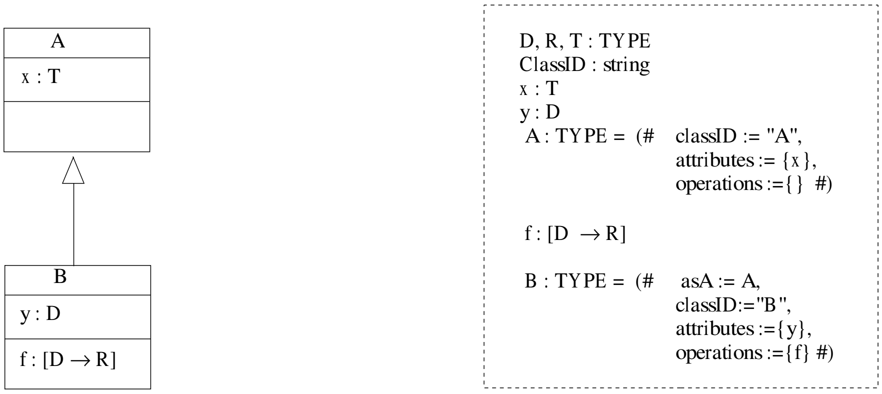


Figure 2: Representation of Inheritance in PVS

3.2 Events and Messages

An Event is a specification of a significant occurrence that has a location in time and space. In a description of communication among system components, we identify three kinds of events: a *local* operation call, a *message send* event, and a *message receive* event. We are interested in externally visible behavior of objects and hence ignore local operation calls. Occurrences of message send and message receive events usually involve invocation of operation of one object by another (not necessarily distinct) object, the *source* and the *target* objects respectively.

Formally, we represent an event as a PVS record type whose fields consist of the event identifier which is identical to the identifier of the associated message, the sender and the receiver objects of the associated message, an attribute that specifies the kind of event, the action that will ensue, and a list of arguments. Symbolically, `Event` type is specified as follows:

```
EventID : TYPE;
Time : TYPE = nat
fin_set[T : TYPE] : TYPE = finite_set[T]
EventKind : TYPE = {send, receive, local}

Event : TYPE = [# eventID    : EventID,
                  sender     : ObjectID,
                  receivers  : fin_set[ObjectID],
                  eventKind  : EventKind,
                  time       : Time,
                  action     : Action #]
```

A message is a specification of a communication among objects, or an object and the environment of the system, and conveys information with the expectation that activity will ensue. It also specifies roles of the sender and receiver objects, as well as the associated action which models the statement that causes the communication to take place. A message can be either a signal (asynchronous) or an operation call (synchronous).

A message may be multi-casted to several target objects. UML, however, does not directly support message broadcasting. Rather, it simulates multicasting by making it possible to target a message to a set of objects. As a result, message receivers are represented as a finite set of objects. Making a distinction between message send events `SendEvent` and message receive events `ReceiveEvent` is necessary to specify behavior of objects participating in the interaction modeled by a sequence diagram. The `SendEvent, ReceiveEvent`, and `LocalEvent` types are specified as predicate subtypes of the `Event` type.

```
e : VAR Event
send?(e) : bool = eventKind(e) = send
recv?(e) : bool = eventKind(e) = receive
local?(e) : bool = eventKind(e) = local

SendEvent : TYPE = (send?)
ReceiveEvent : TYPE = (receive?)
LocalEvent : TYPE  = (local?)
```

In our framework, a message send and the corresponding message receive events are considered to be two distinct instances of event occurrence. A message involves exactly two (not necessarily distinct) objects - the *source*, and the *target*. In case of iterative message passing and message broadcast, each communication is modeled separately. Hence, we model a message as a pair of send and receive events. The correspondence between them has to be established uniquely. The operation to be invoked and its parameters are extracted from the associated action.

An important static constraint on a message is the *causality* requirement which is formalized as a relation between set of `SendEvent` and the set of `ReceiveEvent` - a requirement that guarantees the fact that a message is sent before it is received. The UML supports the notion of time. For a message `m`, `m.sendTime` and `m.receiveTime`, (as described in OMG UML v1.3 [OMG, 1999] pp. 3-98) specify, respectively, the time the message is sent and received. That is the time of occurrences of the associated send and receive events. We capture the notion of time, by stamping every event by the time of its occurrence and to store this information, we adorn the event record with the `time` field. The time information is useful to express temporal properties of traces of events, such as minimum time between occurrences of events. In the sequel, however, we consider only the order of occurrences of events. The global time stamps of events can be used for merging traces by interleaving them in the order of the time of occurrences of events.

3.3 Traces of Events

A trace is a sequence of events that satisfies some predicates on events and program variables such as the causality predicate. The semantics of an object may be described by sets of infinite and finite traces reflecting non-terminating and terminating executions. However, for safety purposes finite trace semantics suffice to specify behavior of a system over a finite time interval, assuming that all iterations terminate, and we consider prefix-closed sets of traces of finite lengths. The PVS library includes a parameterized `list` ADT which is synthesized, during type checking, into a complete theory that specifies the standard `list` data type.

We represent traces of events as a prefix-closed set of finite list of events. To describe essential properties of traces, and ultimately behavior of sequence diagrams they model, we need to define some auxiliary functions on lists and events.

```
t, t1, t2 : VAR list[Event]
prefix(t1, t2) : bool = t1=prefix_upto(length(t1),t2)
```

where the function `prefix_upto()` is a defined below. Note that types and variables that are specified in earlier sections are considered available in later sections and referenced without re-declaration.

```
x, e, e1: VAR Event;     s: VAR setof[Trace];    n : VAR nat
prefix_upto(n,t) : RECURSIVE list[T] =
                       CASES t OF
                             null : null,
                             cons (x, t1) :
                                  IF n = 0  THEN null
                                  ELSE cons(x, prefix_upto(n-1,t1))
                                  ENDIF
                       ENDCASES
                   MEASURE length(t)
```

In PVS, only *total* function calls are allowed, since the domain of function can be restricted by predicate subtyping, termination of all recursive functions must be proved. The `MEASURE` construct is a predefined structure in the PVS specification language and specifies how to prove the termination of recursively defined functions.

```
rank(e,t) : RECURSIVE nat = CASES t OF
                                  null : 0,
                                  cons(x, t1) :
                                      IF x=e THEN 1
                                      ELSE 1 + rank(e,t1)
                                      ENDIF
                                  ENDCASES
                            MEASURE length(t)

prefix_closed(s): bool = s(null) & (∀ e, t: s(cons(e,t)) ⇒ s(t))

es : VAR SendEvent
er : VAR ReceiveEvent
ts, tr : VAR list[Event]

filter_send(er,t) : list[Event] =
                        filter(prefix_upto(rank(er),t), send?)
filter_recv(er,t) : list[Event] =
                        filter(prefix_upto(rank(er),t), recv?)

causal?(t): bool= ∀ er: member(er,t) ⇒
       length(filter_send(er,t))-length(filter_recv(er,t)) >= 0

Trace : TYPE = (causal?)
```

The `prefix()` and `prefix_upto()` functions are used to determine correspondence between send and receive events that may comprise a message. The `filter()` function returns elements of the list, i.e. its first argument, that satisfy the predicate given as the second argument. Note that in the definition of the `rank` function, we are interested in the rank of events that occur in the trace given as an argument. Assigning rank zero to all the events that are not members of the trace does not affect the definition of the causality predicate `causal?`. The type `Trace` contains finite list of events that satisfy the causality predicate.

Next, we define prefix-closure of a given trace `t` and *precedence* relation on the set events w.r.t. a given trace.

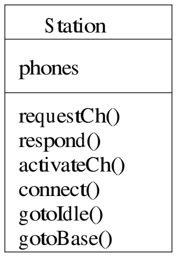
```
n : below(length(t))
prefix_closure(t): setof[Trace]= {prefix_upto(n,t) | true}

precede(e1,e2,t) : bool = rank(e1,t) ≤ rank(e2,t)
```

The `below()` function is predefined in the PVS specification language and returns the set of natural numbers less than or equal to the actual parameter provided.

3.4 The Notions of Classes and Objects

A class describes a set of objects sharing a collection of features, including attributes, operations, and methods. It models the data structure and behavior of its objects. Each object of a class contains its own set of values corresponding to the structural features described in the class. In UML graphical notation, a class is rendered as a rectangular box with three compartments; the topmost compartment for the class name, the middle one for a set of attributes, and the last compartment for a set of operations. An example shown in Figure 3(a) describes a class with name `Station`, attributes `phones`, and operations `requestCh, respond, activateCh, connect, gotoIdle, gotoBase`. Types and initial values of attributes, and signatures of operations, except for the names, are all optional. Figure 3(b) shows a PVS specification of the class meta-model at a higher level of abstraction (details such as the set of interfaces realized by the class are abstracted away), and its instance, the `Station` class. An object is an entity that exhibits observable properties. It specifies an instance of a class on which an operation can be invoked and which has a state that stores the effects of the operations. An object may have a set of attribute values that implement its current state, and is connected to a set of links, where both sets conform to the specification of its class. In UML sequence diagrams, the existence of an object is depicted by an object box and a life-line. A life-line is a vertical line that specifies the existence of an object over a given period of time. Object creation and/or destruction during the interaction specified by the sequence diagram,




```
AttributeLink, Operation, ClassID : TYPE

Class : TYPE =  [#    classID : ClassID,
                      attributes : setof[AttributeLink],
                      operations: setof[Operation],
                      parents : setof[ClassID]  #]

Station : Class =  (#   classID:= station,
                        attributes:= {phones},
                        operations:= {request, ...},
                        parents:= { } #)
```

Figure 3: Representation of a Class in PVS

and ordering of events that may occur on the object are specified. It does not, however, specify the exact time elapsed between occurrences of two events.

The structure of an object is represented by a PVS record whose fields include: an object identifier, a class, a set of attributes, a set of operations, and a set of traces of events that models behavior of the object. Symbolically,

```
AttributeLink : TYPE
ObjectRec : TYPE = [# objectID          : ObjectID,
                      class             : Class,
                      attributeLinks    : fin_set[AttributeLink],
                      traces            : setof[Trace] #]
```

We define the semantics of an object as a prefix-closed set of traces of events or operation calls that satisfy certain properties such as causality. Below, we define, as predicates, requirements that must be fulfilled by elements of type `ObjectRec` to be considered as valid object description. Then, a predicate subtype `Object` of `ObjectRec` that captures semantics of objects is specified.

```
c   : VAR Class;
at : VAR Attribute
op  : VAR Operation;
objr : VAR ObjectRec

classExists?(objr) : bool = NOT empty?(classes(objr))

all_attribs(objr): bool = (∀ at: (slots(objr)(at) ⇒
                   (∃ c: classes(objr)(c) & attributes(c)(at)))
```

```
Object: TYPE = {objr| classExists?(objr) &
                     (∀t: member(t, traces(objr)) ⇒
                     causal?(t) & prefix_closed(traces(objr)))}

classExistLemma : LEMMA (∀ (obj : Object) : classExists?(obj))
```

The functions `attributes` and `operations` return, respectively, the sets of attributes and operations, local and inherited, of a class given as its argument, by recursively traversing its parent classes and interfaces it realizes. The predicates `all_ops`, and `all_attribs` specify that for every operations that may be invoked on an object and for every attribute of the object, there must exist a class in the set of classes of the object in which the operation and the attribute are specified.

In this paradigm where multiple and dynamic classification is supported, i.e. an object can be an instance of several classes, and it may dynamically gain or lose a class during system execution. However, there must always exist at least one class which specifies some structure and behavior of the object. This requirement is stated as the predicate `classExists?` and the lemma `classExistLemma`, where the latter can be discharged by invoking the PVS theorem prover. Other similar requirements such as the conformance of the set of link ends of an object to the set of association ends of one or more of its classes can similarly be stated and proven correct.

4 Semantics of UML Sequence Diagram

Once the basic semantic elements are represented formally, we put them together into a PVS theory that contains representation of the semantic model of sequence diagrams. This approach is in line with the specification style of PVS - an entity should be defined before it can be referenced, and there is no forward reference. The semantic model of a sequence diagram should capture the behaviors that system specified by the sequence diagram should exhibit. For example, invariant properties of the system are stated as *axioms* and *predicates* respectively. Invariants that involve only parts that were separately defined are specified as predicates on the corresponding semantic models.

We represent sequence diagrams, as a PVS record type with fields:

- the identifier of a sequence diagram
- the set of objects participating in the interaction specified by the sequence diagram
- a prefix-closed set of traces of events modeling the interaction. We use a (possibly infinite) set of traces of events in order to capture non-determinism.

In the PVS specification language, a trace can be modeled either as a (possibly infinite) *sequence* or finite *list* of events. The sequence and list data types are predefined in the PVS library. In the sequel, we model traces as lists.

```
SeqDiagrams : THEORY
 BEGIN
 SeqDiagramID: TYPE

 SeqDiagRecord : TYPE = [# seqDiagramID : SeqDiagramID,
                           objects : fin_set[Object],
                           traces : setof[Trace] #]

 sqr : VAR SeqDiagRecord;    obj : VAR Object

 causal(sqr): bool= (∀ t: traces(sqr)(t) ⇒ causal?(t))

 projection : [Trace, setof[Event] → Trace] = filter

 projects(sqr): bool = (∀ obj,t: (traces(sqr)(t) &
                                  objects(sqr)(obj))⇒
         (∀ t1 : traces(obj)(t1) ⇒
            member(projection(t, list2set(t1)), traces(obj))))

 compose(sqr) : bool= (∀ e,t: (traces(sqr)(t) & member(e,t)) ⇒
        (∃ obj: objects(sqr)(obj) ⇒
           member(operation(action(e)), operations(obj))))

 prefix_closed(sqr) : bool = prefix_closed(traces(sqr))

 seqDiag : TYPE = {sqr | causal(sqr) & prefix_closed(sqr) &
                         projects(sqr) & compose(sqr)}
 END SeqDiagrams
```

The `list2set` is a predefined PVS function on `lists` that converts a list into a set. A trace of events is a possible run of the system specified by the sequence diagram if and only if it satisfies the properties specified by the predicates. The `projection` function is defined as the built-in `filter` function and returns projection of a trace on a given set of events. The predicate `projects` states that for every allowable trace of a sequence diagram and an object participating in the interaction specified by the sequence diagram, the projection of the trace onto a trace of the object must be a valid trace of the object. The composition predicate `compose` states that for every event in a valid trace, there must exist an object, in the set of interacting objects, on which the operation associated with the event is invoked. More behaviors, for instance model well-formedness rules, and relationships between elements of sequence diagram can easily be formalized similarly.

5 Case Study: A Mobile Telephone System

5.1 System Description

In this section, we present a case study to demonstrate the use of our approach in rigorous model analysis. Consider a dynamic network of mobile telephone system shown in Figure 4. The network consists of a central telephone exchange `c : Center`, two switching stations `s1, s2 : Station`, and a mobile telephone `p : Phone` attached to a vehicle moving around. This network configuration can be generalized to any finite number of stations and telephones. Each switching station covers a given range of (possibly overlapping) area and the telephone is initially connected to `s1` as shown in Figure 4. Active communication channels are represented as solid lines, whereas inactive channels are represented as broken lines. Before the vehicle moves out of the range of station `s1`, the mobile telephone relinquishes its earlier contact with `s1` and establishes contact with the station `s2`. This scenario is an instance of the notion of dynamic reconfiguration. Our objective is to model the reconnection interaction using UML sequence diagram, encode the model into PVS specification, and formally analyze its correctness and/or consistency with respect to the requirement specification.

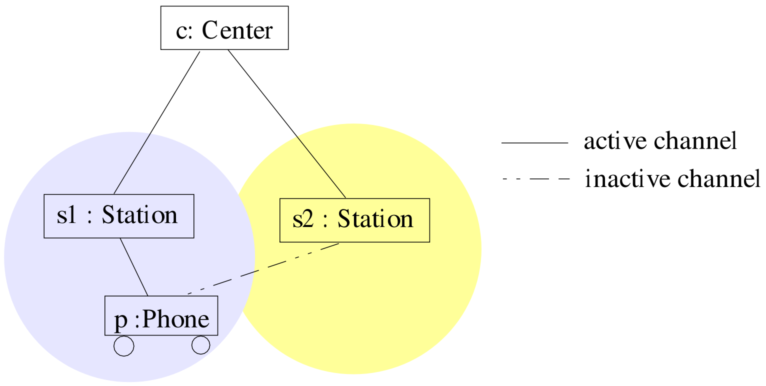


Figure 4: A Mobile Telephone Network

We assume that the switching stations `s1`, and `s2` are permanently connected to the central station `c`, and that the mobile telephone `p` is connected to station `s1` before the interaction begins. A crucial system requirement is that the mobile telephone must remain connected to at least one station at any given time. This is equivalent to saying that, for a mobile telephone the set of *base* stations within its range must remain nonempty. This means that the mobile phone must, at any given time, remain connected at least to one station.

5.2 UML Specification of the System

The class diagram depicted in Figure 5 shows specification of structural properties of the telephone network system described above. The UML sequence

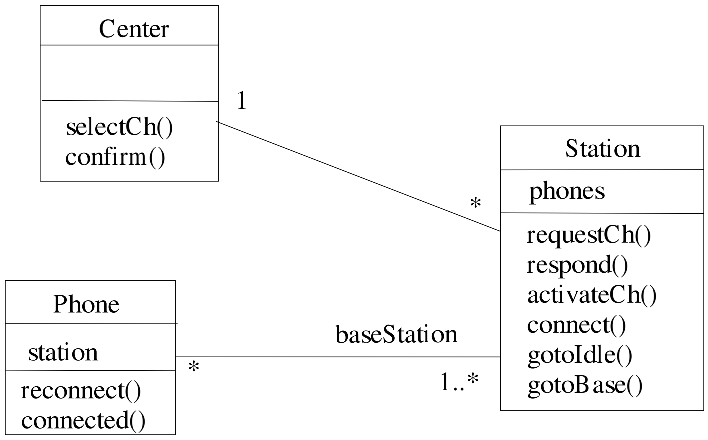


Figure 5: A Class Diagram Specification

diagram shown in Figure 6 models the `reconnection` interaction: when the mobile phone is leaving the range of `s1` and entering the range of `s2`. When the signal from `s1` gets weak, the mobile phone `p` sends a request for a channel to station `s1` which in turn contacts center `c` to get appropriate stations and channels, respectively `s2` and `n` in this case. We assume that `c` is capable, in a way we will not specify, to determine the appropriate station(s) and channel(s). When the station and the channel are confirmed, `c` responds to `s1`. Then, `s1` informs `p` to reconnect to the identified station via the given channel, and `s1` may go to *Idle* state when there is no other telephone connected to it. Finally, `p` establishes a connection to `s2`, and `s2` goes to *base* state.

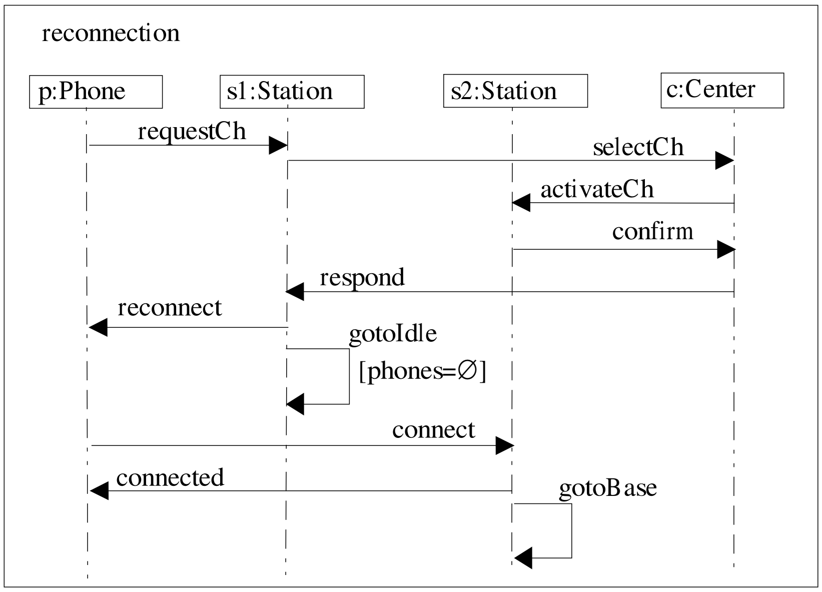


Figure 6: Sequence Diagram: reconnection

5.3 PVS Semantic Models

We provide a fragment of a PVS specification of the interaction described by the sequence diagram shown in Figure 6. The classes `Center`, `Station` and `Phone` are declared as classes with their respective set of attributes and operations (only partially listed in the case of the `Station` class).

```
Operation : TYPE = {requestCh,activateCh,respond,connect,
                    gotoIdle,gotoBase,reconnect,selectCh,confirm}

Attribute: TYPE = {stations: setof[Station],
                   channels : setof[Channel],
                   phones: setof[Phone]}

Center : Class = (#classID := "Center",
                   attributes := {},
                   operations := {selectCh, confirm}
                   asClass := {} #)
```

```
Station : Class = (# classID := "Station", attributes := {phones},
                     operations := {activateCh,respond,connect,
                                    gotoIdle,gotoBase,requestCh},
                     asClass := {} #)

Phone : Class = (# classID := "Phone",
                   attributes : setof[Attribute],
                   operations : {reconnect, connected},
                   traces : prefix_closure((: requestCh,reconnect,
                                              connect,connected:)),
                   parents : { } #)
```

The objects `c, s1, s2` and `p` are declared as an instance of the `Object` type with appropriate values assigned to its fields. We present explicit specification of the objects `p,s1,s2` and `c`. Finally, we sketch an explicit model of the sequence diagram `reconnection`.

```
c, p, s1, s2 : VAR Object
p: Object = (# objectID := "p",
               class := {Phone},
               attributes := {stations} #)
s1 : Object = (# objectID := "s1",
                 classes := {Station},
                 traces:= prefix_closure((: requestCh,selectCh,
                                            respond,reconnect,
                                            gotoIdle:) #)
s2 : Object = (# objectID := "s2",
                 classes := {Station},
                 traces : prefix_closure((:activateCh,confirm,
                                           connect,connected,
                                           gotoBase:)) #)
c : Object = (# objectID := "c",
                classes := {Center},
                attributes := {channels, stations}#)

sq : SeqDiag = (# seqDiagramID := "reconnection",
                  objects := {c, s1, s2, p},
                  traces := {prefix_closure((:p.requestCh,
                                              s1.requestCh,
                                              s1.selectCh,
                                              c.selectCh,...,
                                              s1.gotoIdle,
                                              s2.gotoBase:)),
```

```
                    ⋮
prefix_closure((:p.requestCh, ...
                 s2.gotoBase,
                 s1.gotoIdle:))}#)
```

In description of traces, an event is denoted by the identifier of the object on which the event occurs followed by a dot and the name of the operation to be invoked for `ReceiveEvent` and vise versa for `SendEvent`. For instance, `requestCh.p` is a send event where as `s1.requestCh` is the corresponding receive event.

As mentioned earlier, the specification given in Figure 6, assuming that there is no mobile phone connected to `s1` other than `p`, states that `s1` enters *Idle* state after it sends the `reconnect` message to `p`. Station `s2` becomes a base station for `p` when it receives the `connect` message. The UML sequence diagram shown in Figure 6 does not guarantee that the mobile telephone is connected to the new base station `s2` before station `s1` enters *Idle* state. In the classical message sequence charts (MSC) [ITU-TS, 1996], an approach known as a *general ordering* is used to guarantee deterministic order of event occurrences. UML sequence diagram does not support such an approach and hence a need for formal semantics that ensure this sort of behavior of systems.

Once a UML sequence diagram modeling a system interaction is encoded into PVS specification language as a prefix-closed set of traces of events, temporal properties of the system can be stated as predicate on the traces. For instance, the `idlePred` predicate given below constrains the station object `s1` from becoming `Idle` before the mobile phone is reconnected to a new base station `s2`.

```
idlePred(t:Trace): bool =
        (∀ t, sq: traces(sq)(t): precede(connected,gotoIdle))
pv : VAR Phone; sv : VAR StationID;
cv : VAR Center; chv : VAR Channel

isConnectedTo(pv,sv): bool= attributes(sv)(pv)&attributes(pv)(sv)

mayConnectTo(pv,sv): bool= (∃ cv: attributes(cv)(sv) &
                                  NOT attributes(pv)(sv))

connectivityPred(pv): bool = attributes(pv)(stations) ≠ ∅

theorem1 : THEOREM (∀ sv, pv:
           NOT (isConnectedTo(pv,sv) & mayConnectTo(pv,sv)))
```

System requirements are stated as theorems, and we verify that a specification meets the requirements, we need to discharged the theorems using the PVS proof system. For instance, the theorem `theorem1` captures the fact that a mobile telephone is either connected or not connected to a station, but not both. The theorem can be discharged automatically by a single prover command *"grind"*. The following is a snapshot of a proof of the theorem. `theorem1`:

|-------
{1} $\forall$ (pv, sv: Class):

$\neg$ (isConnected(pv, sv) & mayConnectTo(pv, sv))

Skolemizing,
`theorem1`:

{-1} (isConnected(pv′, sv′) & mayConnectTo(pv′, sv′))
|-------

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `theorem1`.
Q.E.D.

Although the theorem follows straightforwardly from the definitions given above, it clearly demonstrates how the integrated framework enables us to exploit the strengths of the UML notations and the PVS proof system in requirement engineering. The UML models enable us to describe systems at appropriate level of abstraction to improve our understanding of the system in question. They can be used as contract between the stakeholder. The corresponding semantic models that are obtained by translating the UML models into PVS specification language, augmented with additional PVS expressions if need be, enable us to verify important system requirements.

Two points are worth discussing in connection with the translation of UML sequence diagrams into PVS, and the integration of UML CASE tools and the PVS toolkit. Firstly, we discuss how the semantic models resulting from translation of graphical UML models and the PVS proof system interact. The semantic models may not be sufficient to capture system requirements that would be verified, and hence it may be necessary to augment them with pure PVS expressions. Verification of the overall system requirements by using the PVS proof system is straightforward as the whole system specification is in PVS. A drawback of this approach is that users that may not be experts in formal methods should directly deal with formal specifications on PVS side. This contradicts our aim of hiding formal artifacts at the back-end so that users interact with the graphical front-end. An alternative approach is to specify the additional requirements in an ad hoc language such as the object constraint language (OCL)

[Warmer and Kleppe, 1999] and translate the OCL expressions into PVS language, and reason about the constraints using the PVS theorem prover.

Secondly, the integration of a UML CASE tool and the PVS toolkit into a single platform requires a mapping of semantic models into the corresponding UML models. For instance, if the PVS theorem prover detects an error in the PVS semantic model during a verification process, how can this be communicated to users that are not experts in PVS? This can be done by developing a browser that *reverse engineer* the translation of UML into PVS. Keeping records of correspondence between UML modeling elements and their counterparts in PVS specifications simplifies the parsing. For instance, by using the same identifers in UML models and the corresponding PVS semantic models will significantly simplify propagation of errors detected during verification onto the UML models. This is, however, out of the scope of this papers and one of the potential issues for future work.

6 Conclusion and Future Work

In this paper we outline a framework for formalization of UML constructs. Expressing semantic models of UML constructs in a formal specification language enables us to rigorously analyze the models. The resulting semantic models are amenable to rigorous analysis, and facilitate the design and implementation stages as well as use of formal techniques in software verification and validation tasks. Moreover, the underlying formal language and its tool set is used to underpin CASE tools that are developed to automate model analysis. In our case, once the UML modeling constructs are translated into semantic model in PVS-SL, general properties of UML models, such as well-formedness rules, can be stated and proved correct by using PVS tools like theorem-prover and typechecker. The PVS theorem prover discharges most of the proof obligations with little interaction from the user if the requirements are well formulated - and not involving complex quantifier reasoning.

This work 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. It is a part of our long-term vision to explore how the PVS tool set could be used to underpin practical tools to analyze UML models.

There are several related research works on the formalization of UML constructs in the literature [Shroff and France, 1997, Evans, 1998, Evans et al., 1998, France et al., 1998, Whittle, 2000] mostly using `Z` [Spivey, 1992] as the underlying semantic foundation. The work on encoding of CSP [Hoare, 1985] in PVS [Dutertre and Schneider, 1997], is similar to ours. A distinguishing feature of our work is the integration of informal graphical modeling notations and highly expressive formal notations, and utilization of existing tools to analyze UML models. For relevant and detailed information, the reader may refer to our earlier works on formalization of other UML modeling techniques: structural modeling techniques [Aredo et al., 1999, Aredo, 1999], and state machines [Traore, 2000].

A UML sequence diagram describes a fragment of dynamic system behavior resulting in a partial specification. To achieve a more complete system description, one needs to combine several models such as class and statechart diagrams, i.e. different *viewpoints* in UML vocabulary. When different modeling languages are combined, their relationship should clearly be defined, and consistency between different viewpoints must be maintained. In the future, we will investigate how different UML modeling constructs can be used in combination and how they complement each other without violating consistency. Model checking will also be among the research topics we will investigate in the future. Reverse engineering of PVS semantic models to UML models is among topics for future investigation.

Acknowledgements

I would like to thank Olaf Owe, Wenhui Zhang, and Issa Traore for fruitful discussions and comments. This work was financed by the Research Council of Norway (NFR) through the research program for Distributed IT-Systems. Comments by the anonymous reviewers were useful for the improved presentation of this paper.

References

[Aredo et al., 1999] Aredo, D., Traore, I., and Stølen, K. (1999). An Outline of PVS Semantics for UML Class Diagrams (extended abstract. In *the Proc. of The 11th Nordic Workshop on Programming Theory NWPT'99*, Uppsala, Sweden.

[Aredo, 1999] Aredo, D. B. (1999). Formalization of UML class Diagrams in PVS (Extended Abstract). In *the Proc. of Workshop on Rigorous Modeling and Analysis with the UML: Challenges and Limitations, at OOPSLA99.*, Denver, Colorado, USA.

[Blair and Blair, 1999] Blair, L. and Blair, G. (1999). Composition in Multi-Paradigm Specification Techniques. In *the Proc. of 3rd International Workshop on Formal Methods for Open Object-based Distributed Systems (FMOODS'99)*, Florence, Italy. Kluwer.

[Booch et al., 1999] Booch, G., Rumbaugh, J., and Jacobson, I. (1999). *The Unified Modeling Language User Guide*. Addison Wesley Longman Inc, Reading Massachusetts 01867.

[Breu et al., 1997] Breu, R., Grosu, R., Hofmann, C., Huber, F., Kruger, I., Rumpe, B., Schmidt, M., and Schwerin, W. (1997). Exemplary and Complete Object Interaction Descriptions. In Kilov, H., Rumpe, B., and Simmonds, I., editors, *the Proc. of OOPSLA'97 Workshop on Object-oriented Behavioral Semantics*, Atlanta, Georgia. TUM-I9737.

[Crow et al., 1995] Crow, J., Owre, S., Rushby, J., Shankar, N., and Srivas, M. (95). A Tutorial Introduction to PVS. In *WIFT'95: Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, Florida, USA.

[Damm and Harel, 1999] Damm, W. and Harel, D. (1999). LSC's: Breathing Life into Message Sequence Charts. In *Formal Methods for Open Distributed Systems (FMOODS'99)*, Florence, Italy.

[Dutertre and Schneider, 1997] Dutertre, B. and Schneider, S. (1997). Embedding CSP in PVS: An Application to Authentication Protocols. In *Theorem Proving in Higher Order Logics: 10th International Conference, TPHOLs '97*, volume 1275, Murray Hill, NJ. Springer-Verlag.

[Evans, 1998] Evans, A. (1998). Reasoning with UML Class Diagrams. In *the Proc. of WIFT'98*. IEEE Press.

[Evans et al., 1998] Evans, A., France, R. B., Lano, K., and Rumpe, B. (1998). Developing the UML as a formal modelling notation. In Bézivin, J. and Muller, P.-A., editors, *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998*, pages 297–307.

[France et al., 1997] France, R. B., Bruel, J.-M., and Larrondo-Petrie, M. M. (1997). An Integrated Object-Oriented and Formal Modeling Environment. *Journal of Object-Oriented Programming (JOOP)*, 10(7).

[France et al., 1998] France, R. B., Evans, A., Lano, K., and Rumpe, B. (1998). The UML as a Formal Modeling Notation. *Computer Standards & Interfaces*, 19:325–334.

[Gunter et al., 2001] Gunter, E. L., Muscholl, A., and Peled, D. A. (2001). Compositional Message Sequence Charts. In *the Proc. of TACAS 2001*. Springer-Verlag Heidelberg. LNCS 2031.

[Haugen, 1997] Haugen, O. (1997). *Practitioners Verification of SDL Systems*. PhD thesis, University of Oslo.

[Hoare, 1985] Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Prentice Hall.

[ITU-TS, 1996] ITU-TS (1996). ITU-TS Recommendation Z.120: Message Sequence Chart (MSC).

[Kammüller and Helke, 2000] Kammüller, F. and Helke, S. (2000). Mechanical Analysis of UML State Machines and Class Diagrams. In *the Proc. of Workshop on Precise Semantics for the UML. ECOOP2000*, Cannes.

[OMG, 1999] OMG, T. (1999). OMG Unified Modeling Language Specification, version 1.3. OMG standard document.

[Owre et al., 1999a] Owre, S., Shankar, N., Rushby, J., and Stringer-Calvert, D. W. (1999a). PVS Language Reference, version 2.3. Computer Science Laboratory.

[Owre et al., 1999b] Owre, S., Shankar, N., Rushby, J., and Stringer-Calvert, D. W. (1999b). PVS System Guide, version 2.3.

[Owre et al., 1993] Owre, S., shankar, N., and Rushby, J. M. (1993). The PVS Specification Language. Computer Science Lab., SRI International.

[Reggio et al., 2000] Reggio, G., Astesiano, E., Choppy, C., and Hussmann, H. (2000). Analysing UML Active Classes and Associated State Machines – A Lightweight Formal Approach. In Maibaum, T., editor, *the Proc. Fundamental Approaches to Software Engineering (FASE 2000), Berlin, Germany*, volume 1783 of *LNCS*. Springer.

[Rumbaugh et al., 1999] Rumbaugh, J., Jacobson, I., and Booch, G. (1999). *The Umified Modeling Language, Reference Manual*. Addison Wesley Longman Inc., Reading Massachusetts 01867.

[Shroff and France, 1997] Shroff, M. and France, R. B. (1997). Towards a formalization of UML Class Structures in Z. In *the Proc. of the COMPSAC'97*.

[Spivey, 1992] Spivey, J. M. (1992). The Z Notation: A Reference Manual.

[Traore, 2000] Traore, I. (2000). An Outline of PVS Semantics for UML Statecharts. *Journal of Universal Computer Science*, 6(11).

[Warmer and Kleppe, 1999] Warmer, J. B. and Kleppe, A. G. (1999). *The Object Constraint Language: Precise Modeling with UML*. Addison Wesley Longman Inc., Reading Massachusetts 01867.

[Whittle, 2000] Whittle, J. (2000). Formal Approach to Systems Analysis Using UML: An Overview. *Journal of Database Management*, 11(4).