Journal of Universal Computer Science, vol. 6, no. 11 (2000), 1088-1108
submitted: 16/6/00, accepted: 19/9/00, appeared: 28/11/00 © Springer Pub. Co.

An Outline of PVS Semantics for UML Statecharts

Issa Traoré
(University of Victoria
Department of Electrical and
Computer Engineering
PO Box 3055 STN CSC
Victoria BC V8W 3P6
Canada
itraore@ece.uvic.ca)

Abstract: The current UML standard provides definitions for the semantics of its components. These definitions focus mainly on the static structure of UML, but they don't include an execution semantics. These definitions include several "semantic variation points" leaving out the door open for multiple interpretations of the concepts involved. This situation can be handled by formalizing the semantic concepts involved. In this paper we present an approach for the formalization of one of the multiple diagrams of UML, namely statechart diagrams. That is achieved by using the PVS Specification Language as formal semantics domain. We present also how the approach can be used to conduct a formal analysis using the PVS model-checker.

Key Words: Open Distributed Systems, Formal Methods, Object-orientation, UML, PVS, Specification.
Categories: D.1.5, D.2.4, D.3.1

1 Introduction and Problem Statement

The increasing development of both local and wide area networking has made distributed processing one of the most important topics in computing. More specifically, standardisation effort have been achieved in order to enable open distribution [ISO-rmodp]. Among the new challenges, there is the ability to integrate formal methods in the development cycle of open distributed systems. Most of the techniques supporting the development of open distributed systems, such as UML [OMG 1999], lack the formal semantics and mechanisms underlying formal development methods. On the other hand, few of the existing formal methods can be fitted well to open distributed systems. For instance in the RM-ODP documents [ISO-rmodp], several formal languages such as LOTOS [ISO-lotos 1988], Z [Spivey 1989], SDL [ISO-sdl 1993] and Estelle [ISO-estelle 1989], are proposed for the description of the various viewpoints involved. But as pointed out in [Dahl 1998], these languages are only partly satisfactory. For instance one may use Z for the description of the static parts of the information viewpoint, but Z is not well-suited to deal with the dynamic aspects. Moreover, SDL and Estelle give little support for formal verification. Finally, due to its rather operational nature, LOTOS is mainly suited for the design stage.

Hence, taking the previous remarks into account, one obvious solution is to build-up a completely new method from scratch. However, this is extremely costly and as mentionned in [Abadi 1994], "it would be unfortunate if every new class of systems required inventing new semantics, along with proof rules, languages, and tools". An alternative would be to adapt and combine already existing technologies, more specifically graphical object-oriented notations, which are semi-formal and formal notations, which provide a rigorous semantic foundation.

Several works have attempted to provide a mathematical basis for the concepts underlying object-oriented models. Some of these approaches consist of adapting or extending a novel or existing formal description technique with object-oriented concepts [Moreira 1994]. Others derive a formal specification from the semi-formal (or informal) model built with existing object-oriented notations such as UML or OMT [Hayes 1991]. The main problem with these approaches is that the user should have to deal with a certain amount of formal artifacts, and that can be a barrier to an industrial use. A third approach that has been adopted in this work consists of assigning a formal semantics to an existing object-oriented notation [Evans 1998]. In this case, the formal "stuff" is hidden behind the graphical one. The user deals with the graphical model, while the formal "stuff" is processed automatically at the back-end.

We have decided to base our work on UML as an industrial-strength specification language and the PVS Specification Language (PVS-SL) [Owre *et al.* 1995], which provides corresponding formal semantics. In previous work we dealt with the formal semantics of UML class diagrams in PVS-SL [Demissie *et al.* 1999]. This paper focuses more specifically on the formal semantics of UML statecharts, which represents with class diagrams one of the most important diagrams of UML. Our objective in this work is twofold:

1. Produce a precise specification of the system by removing any ambiguity from the model.
2. Analyse the model by ensuring that any valid implementation of the system guarantees the specification.

Our first goal is achieved by providing in PVS-SL an abstract syntax for the statechart diagram, and by generating and checking a set of corresponding well-formedness rules. The second goal is achieved by defining a semantics model for the diagram expressed in PVS-SL, and by checking the properties of the system against this model. That is done using the model-checking and proof-checking capability of the PVS toolkit. However, as mentioned in [Evans *et al.* 1999], understanding and formulating the need for proof is outside the experience of most developers. Furthermore, most of them "will not, in the foreseeable future, be willing to use abstract formal languages and notations to design software, regardless of how theoretically desirable it might be to do so." Therefore, the only way to get developers to be more precise is to provide tools that will help efficiently in doing so. For this reason, we have decided to target model-checking as our main analysis technique. Theorem proving may be used as a complementary solution and only for specific applications. This allows us to provide a verification process that is fully automated.

The next section is a short overview of the notations involved in the framework. Then, we present our formalization approach and few results. Finally, in the subsequent sections, we develop a case study based on the requirements of an elevator system, and make some concluding remarks. A complete list of the definitions involved in the approach is given in [Traoré 2000].

2 Presentation of the Notations involved

UML is built on an object-oriented framework; therefore, it provides several capabilities such as *encapsulation, data abstraction, extensibility, reusability* and *flexibility*, which are useful for the description of open distributed systems. It has been standardized and there are many tools available on the market today. In addition, UML provides an underlying methodology for specification and refinement, and a diagramatic notation which contributes to communicativeness and friendliness.

PVS-SL is used in this platform, as semantics foundation and not as a specification language. As a result, the user will not need to have an in-depth knowledge of the PVS formal notation and proof system. PVS-SL offers a very general semantic foundation and a set of powerful tools. It is highly expressive and offers several mechanisms for formal analysis. Compared to OCL, PVS-SL provides stronger support for all kinds of operations (with or without side-effects). For instance, in OCL, operations can be modeled by a recursive expression, but it is the modeler's responsibility to ensure the well-definedness of the recursion. In PVS-SL, however, this is handled by the built-in construct: the *MEASURE* clause.

2.1 More About UML Statecharts

2.1.1 Comparison with Classical Statechart:

UML statechart is an object-based variant of Harel statecharts. It includes several concepts derived from ROOMcharts, a variant of statechart defined in the ROOM language. A statechart diagram is used to describe the behavior of model element such as an object or an interaction. It consists of possible sequences of states and actions through which the component may proceed during its lifetime. The behavior of a component is described by specifying reaction to discrete events such as signals and operation calls.

The main difference between UML statechart and Harel statechart is that the former represents behavior of a type whereas the latter specify behaviors of processes: actually in UML statecharts the notion of process is not supported. In UML statecharts, events are not considered as primitive signals since their definition may include parameters. Event conjunction is not supported: the semantics is not given with respect to a general system context; instead, only single event dispatch is considered. Operations are not broadcast; instead, they can be directed to an object-set. However, the notion of synchronous communication between state machines is supported. Another difference worth mentioning is that whereas in classical statechart the order of execution of actions associated to transitions doesn't matter, in UML statecharts, they are executed in their given order. It is also important to note that in classical statecharts, the execution of transitions takes zero time. The execution of the whole system is based on synchronous steps; each step produces new events which are dealt with at the following step. In contrast, UML statecharts are based on the software execution model based on threads of execution and the assumption that execution may take time.

2.1.2 Overview of the notation:

We give in the following an overview of the UML statechart notation through an example statechart diagram for an elevator system depicted in figure 1.

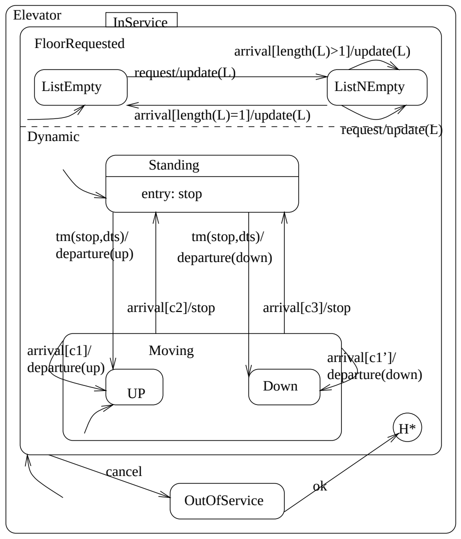

Figure 1: Statechart Diagram of the Elevator System

A statechart diagram is a graph representing a state machine. The key components of the graph are transitions, states and various other types of vertices (pseudostates). States are rendered as a rectangle with rounded corners. Pseudostates are rendered by appropriate pseudostate symbol. A state can be either a simple state or a composite state. A simple state is a state that cannot be further refined (i.e. that has no substate). A composite state is a state that can be decomposed into two or more concurrent substates or into mutually exclusive disjoint substates (also called a sequential state). Every statechart diagram has a root state that contains all the other elements of the entire graph.

In the diagram of figure 1, the root state is state *Elevator*, which consists of two direct substates, namely, state *InService* and state *OutOfService*. State *OutOfService* is a simple state whereas state *InService* is a concurrent composite state. State *InService* has two direct substates (called also regions), namely, states *Dynamic* and *FloorRequested*, which are composite sequential states by definition, decomposed in their turn into mutually exclusive disjoint substates. A pseudostate is an abstraction used to connect multiple transitions into more complex state transitions paths. It consists of various kinds, more specifically *initial, deepHistory, shallowHistory, join vertices, fork vertices, junction vertices* and *choice vertices*. *ShallowHistory* and *deepHistory* which are some of the most used, are shorthand notations used to depict historical information. *Deep-*

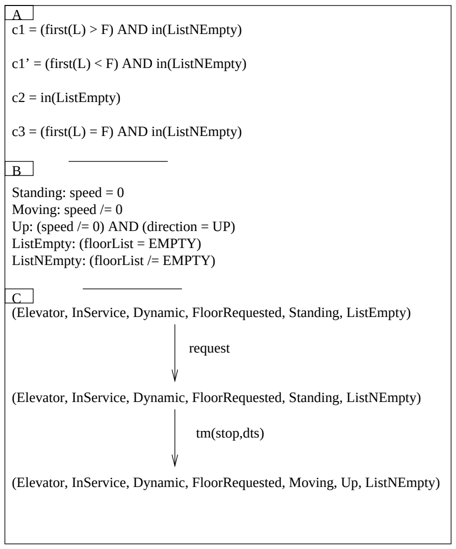
A
c1 = (first(L) > F) AND in(ListNEmpty)
c1' = (first(L) < F) AND in(ListNEmpty)
c2 = in(ListEmpty)
c3 = (first(L) = F) AND in(ListNEmpty)
B
Standing: speed = 0
Moving: speed /= 0
Up: (speed /= 0) AND (direction = UP)
ListEmpty: (floorList = EMPTY)
ListNEmpty: (floorList /= EMPTY)
C
(Elevator, InService, Dynamic, FloorRequested, Standing, ListEmpty)
request
(Elevator, InService, Dynamic, FloorRequested, Standing, ListNEmpty)
tm(stop,dts)
(Elevator, InService, Dynamic, FloorRequested, Moving, Up, ListNEmpty)

Figure 2: Conditions, state predicates and configurations

History represents the most recent active configuration of the composite state that directly contains it. *ShallowHistory* represents the most recent active direct substate of its containing state. For instance, state *InService* contains a deep history state for which the notation is a circle with an H^* inside.

Transitions are rendered by directed arcs interconnecting the states involved. A transition is a relationship between two states, describing the fact that an object in the first state may evolve to the second state and perform specific actions, at the occurrence of a specified event and under specific conditions. A transition may be labeled by a *transition string* using the following format: $event \Leftrightarrow signature[guard \Leftrightarrow condition]/action \Leftrightarrow expression$. Several instances of transitions are provided by the graph in figure 1. For example, the transition labelled *tm(stop,dts)[c1]/departure(up)*, which connects states *Standing* and *Moving*, means that at the occurrence of event *tm(stop,dts)*, and if the condition *c1* is true, state *Standing* is exited, whereas state *Moving* is entered, and action *departure(up)* is executed.

An event represents a noteworthy occurrence. There are various kinds of events (not necessarily mutually exclusive), namely *change event, signal event, call event* and *time event*. A *change event* represents a designated condition becoming true; a *signal event* describes an explicit signal received from one object to another; a *call event* corresponds to the receipt of a call for an operation by an object; a *TimeEvent* corresponds to the passage of a designated period of time after a designated event. For instance, event *tm(stop,dts)* is a time event that occurs *dts* time duration after the occurrence of event *stop*.

2.2 Short Overview of PVS-SL

PVS is a Prototype Verification System for formal specification and reasoning. It consists of a specification language, a parser, a typechecker, a prover, a model-checker, specification libraries and several browsing tools. A PVS specification consists of a collection of theories. A theory consists of type and constant definitions, and related axioms, definitions and theorems. Parametric theories, using types and values, are supported. The language is based on simply typed higher-order logic. Types can be defined using base types (booleans, numbers, etc.), functions, record, and tuple construction. Unintepreted base types and predicate subtype of a given type are some of the significant enhancements to the type system.

```
bag[item: TYPE+]: THEORY
 BEGIN
  bag: TYPE+
  b: VAR bag
  i: VAR item
  empty: bag
  nonemptybag? (b): bool = b /= empty

 put: [item, bag -> (nonemptybag?)]
 get: [(nonemptybag?) -> item]

 empty_ax: AXIOM
               get(put(i,empty)) = empty

END bag
```

Figure 3: Example of PVS Theory

We give in figure 3, an instance of parametric theory called *bag*. The theory receives as parameter a type named *item* , which corresponds to the type of the items contained by the bag. *THEORY, BEGIN* and *END* are the specific keywords that define a theory. Keywords TYPE, VAR, AXIOM and *THEOREM* are used to define respectively type, variable, axioms and theorems. For instance, the axiom labelled *empty_ax* states that the application to an empty bag of a *put* function, followed by a *get* function will give rise to the same empty bag.

2.2.1 The PVS Model-checker:

The model checker of PVS is based on the branching time temporal logic called Computation Tree Logic (CTL) [Shankar 1997]. CTL formulas consist of atomic propositions, propositional combinations, universal path quantified formulas (denoted by **AG***f* — "along every computation path, always *f*", **AF***f* — "along every path, eventually *f*"), existential path quantified formulas (**EG***f* — "along some path always *f*", **EF***f* — "along some path eventually *f*"). In the PVS model-checker, an invariant property in CTL is specified in the form $AG(trans, prop)(s)$

where *trans* is the transition starting state. The AG operator then means that the property *prop* must be true of all states that can be reached from s by the transitions of *trans*. The concept of state considered here is that of computation state which is equivalent to the notion of global state defined later. The notion of computations over a computation state is specified by a parametric theory *state*, provided in the PVS prelude, which receives a state (e.g. computation or global state) as parameter (see figure 4).

```
state[state: TYPE]: THEORY
 BEGIN
  IMPORTING sequences[state]

   statepred: TYPE = PRED[state]  %assertions
   Action: TYPE = PRED[[state, state]]  % transition relation
   computation: TYPE = sequence[state]

  pp: VAR statepred
  action: VAR Action
  aa, bb, cc: VAR computation

  Init(pp)(aa): bool = pp(aa(0))

       ...
END state
```

Figure 4: Theory state

A program represents a set of computations and can be characterized by an initialization assertion (defining the initial state) and a binary transition relation (e.g $Action$) which constrains the allowable transitions of a computation. A computation is defined as a sequence of states and an assertion is modelled as a predicate on states.

3 Formalization Approach

3.1 Overview of the General Approach

Our general approach consists of the following steps:

1. The UML statechart diagram is converted in an abstract syntax using PVS-SL.
2. Then on the basis of that syntax which is specific to the model considered, a set of well-formedness rules are provided under the form of PVS theorems.
3. The rules are checked and the model is updated accordingly.
4. Next, the formal semantics of the model in the PVS specification language is generated. The semantics model obtained is a combination of generic formulas and additional formulas provided by the user in order to complete the definitions of specific model features such as elementary states and events.
5. The semantics model obtained will serve as basis for the verification of the model using the PVS model-checker and the system properties specified by the user.

The whole formal model defined for a given UML statechart diagram consists of three generic PVS theories each describing different aspect of the model, namely, the abstract syntax, the set of well-formedness rules and the formal semantics, and called respectively *AbstractSyntax, WellFormedness* and *FormalSemantics*. Theories *WellFormedness* and *Semantics* are parametric theories which receive as actual parameter the abstract representation of the statechart. The key abstraction of theory *AbstractSyntax* is the formal representation of a statechart diagram which is given as a record listing the different collections of features (e.g. states, events, transitions etc.) involved in the specific diagram considered. From this definition, we can derive a collection of finite sets representing the basic building blocks of the statechart (e.g. set of states, set of events etc.), that actually describe the domain over which the well-formedness rules provided by theory *WellFormedness* are defined. Since this domain is finite, all the well-formedness rules involved can be checked algorithmically. Theory *FormalSemantics* provides input to the PVS model checker, by defining the features involved in the formal semantics of the statechart. More specifically the formal semantics of a statechart is defined as a triple *(globalState, InitState, NEXT_step)*, where *globalState* represents the global states involved, *InitState* is the initial global state, and *NEXT_step* is the transition relation.

3.2 Basic Concepts and Abstract Representation of a Statechart Diagram

The foundation of our formalization activity consists of defining a set of elementary predicates that describe relevant properties about the system state or the system operation. The set of elementary predicates is then partitioned into elementary states and events. A state describes a condition of the system that has a nonnull duration. The state of the whole system at a given time, identified by the term *global state*, is the conjunction of all elementary states that hold during that time. An event describes conditions that can hold only at a particular instant of time.

Another key aspect of our formalization approach is the fact that time is explicitly taken into account as a system variable; this allows us to state time constraints between the occurrence of different events. Time is represented by a single time value for events, which are instantaneous, and a pair of time values for states, where the pair denotes the time interval during which the state holds. Time intervals associated with state predicates are, by convention, closed at the left and open at the right (e.g $[t1, t2[$). No assumptions are made in the UML informal semantics about the time intervals involved in event communication; this leaves open the possibility of different semantics models. Hence, our semantics model will assume *zero decision times*. We define these features in the PVS Specification Language by providing appropriate type definitions.

```
AbstractSyntax : THEORY
                 BEGIN
                   V: TYPE
                   Time: TYPE FROM nat
                   Event: TYPE
                   Vertex: TYPE
                   State: TYPE FROM Vertex
```

```
        defer(s: State): set[Event]
```

We make a distinction between type *State* defined for elementary states and type *state* defined in the PVS prelude for computation states. *V* represents an uninterpreted datatype, which describes the instance variables involved in the feature described by the model. For a class, that consists of the attributes and relationships involved. It may be provided under the form of a record whose fields correspond to the instance variables involved. We define a function *defer*, which associates to a state a set of deferred event. A deferred event is retained until the state machine reaches a state configuration where it is no longer deferred.

Actually, we define a state as a subcategory of a state vertex, which is an abstraction of a node in a statechart graph. A state vertex can be one of the following four subcategories: state, pseudostate, synch state and stub state. A synch state is a vertex used for synchronizing the concurrent regions of a state machine. A stub state is a state vertex used to reference a state machine within another state machine. Hence we give the following definition:

```
PseudoState: TYPE FROM Vertex
SynchState: TYPE FROM vertex
StubState: TYPE FROM Vertex
ax_vertex: AXIOM
        FORALL (x: Vertex):
                PseudoState_pred(x) OR SynchState_pred(x) OR
                StubState_pred(x) OR State_pred(x)
```

Given a state, we define the set of state vertices (possibly empty) directly contained by the state, by providing a function called *dsubvertex* which defines a tree structure on the set of states associated to a state machine. Using *dsubvertex*, we define recursively *subvertex* as the set of all the state vertices contained in a given state.

```
s: VAR State
x: VAR Vertex
dsubvertex(s): set[Vertex]
ax_dsubv: AXIOM
        is_finite(dsubvertex(s)) AND
        singleton?( { s | member(s',dsubvertex(s)) }
subvertex(s): RECURSIVE set[Vertex] =
                        union(dsubvertex(s),
                        { x: (subvertex(y)) | member(y,dsubvertex(s)) } )
            MEASURE(LAMBDA(s): dsubvertexs /= emptyset)
subvertex1(s): set[Vertex] =
            { Y: State | y=s OR member(y,subvertex(s)) }
```

subvertex1 provides a set including in addition to the subvertices of a state, the state itself.

Composite states and simple states are defined by providing appropriate predicates based on the previous definition:

```
compositeState?(s): bool = subvertex(s) /= emptyset
simpleState?(s): bool = subvertex(s) = emptyset
```

We define also a function called *container*, which returns the state directly containing a given state vertex.

```
container(x): State = s WHERE
                        (compositeState?(s) AND member(x,dsubvertex(s)))
```

We deduce from the previous definitions the set of substates and the set of direct substates of a given state, as follow:

```
substate(s): set[State] = { x | member(x, subvertex(s)) AND State_pred(x) }
dsubstate(s): set[State] = { x | member(x, substate(s)) AND container(x) = s }
```

Next we define a transition as a record whose fields correspond to its main features (e.g. trigger, guard condition etc.).

```
Action: TYPE+
Condition: TYPE = pred[V]
Context: TYPE = { behavioral, classifier }
Transition: TYPE = [# source: Vertex,
                      trigger: Event,
                      guard: Condition,
                      effect: Action,
                      target: Vertex # ]
```

Context is an enumerated type that defines the kind of model element being described by the statechart (either a behavioral feature or a classifier).

Finally, we give an abstract representation for a statechart diagram itself by defining a type named *StateMachine* as a record whose fields correspond to the collections of items involved in the diagram.

```
Statemachine: TYPE = [#
                      State: set[State],
                      StubState: set[StubState],
                      SynchState: set[SynchState],
                      Initial: set[Initial],
                      Choice: set[Choice],
                      DeepH: set[DeepH],
                      ShallowH: set[ShallowH],
                      Join: set[Join],
                      Fork: set[Fork],
                      Junction: set[Junction],
                      PseudoState: { v: set[PseudoState] | v=union(DeepH,
                                     union(ShallowH,union(Choice,
                                     union(Join,union(Fork,
                                     union(Junction,Initial)))))) },
                      Vertex: { v: set[Vertex] |
                                     v =union(PseudoState,
                                     union(StubState,
                                     union(SynchState,State))) },
                      CallEvent: set[CallEvent],
                      TimeEvent: set[TimeEvent],
                      ChangeEvent: set[ChangeEvent],
SignalEvent: set[SignalEvent]
Event: { E:set[Event] |
                    E=union(CallEvent,
                    union(TimeEvent,
                    union(ChangeEvent,
                    SignalEvent))) },
Action: set[Action],
Condition: set[Condition],
Transition: { T: set[Transition] |
                    FORALL (tr: (T)):
                    member(source(tr),Vertex) AND
                    member(target(tr),Vertex) AND
                    member(trigger,Event) AND
                    member(guard,Condition) AND
                    member(effect,Action) },
Root: { s: (State) |
                    compositeState?(s) AND
                    (subvertex1(s) = Vertex) },
Context: Context
# ]
```

3.3 Well-Formed Diagram

Well-formedness rules are defined within a parametric theory called *WellFormedness*, which receives as parameter the abstract representation of a specific statechart diagram. We have identified seven rules provided under the form of PVS theorems. The complete listing of the theory can be found in [Traoré 2000]. Before presenting in the following some of these rules, we define some auxiliary functions and introduce the concept of state configuration. A state configuration is defined as a maximal set of simultaneous active states.

```
WellFormedness [ sm: StateMachine] : THEORY
                   BEGIN
                     IMPORTING AbstractSyntax
                     x: VAR Vertex
                     atleast2?(A: finite_set): bool = A /= emptyset AND
                                                      NOT singleton?(A)
                     atmostst1?(A: finite_set): bool = A = emptyset OR
                                                      singleton?(A)
                     incoming(x): set[Transition] = { tr: (Transition(sm)) |
                                                      x = target(tr) }
                     outgoing(x): setof[Transition] = { tr: (Transition(sm)) |
                                                      x = source(tr) }
                     configuration: TYPE = {C: set[State] |
                                subset?(C,State(sm)) AND
                                member(Root(sm),C) AND
                                (FORALL (s:(C) | compositeState?(s)):
                                IF isConcurrent(s)
                                THEN subset?(dsubstate(s),C)
                                ELSE
```

```
                         singleton?(intersection(dsubstate(s),C))
                         ENDIF)
                         }
```

The first rule considered, labelled *wf1*, is related to composite states:

```
wf1: THEOREM
    (FORALL (S: (State(sm)), R: (State(sm)) |
             compositeState?(S) AND compositeState?(R)):
             atmost1?(intersection(Initial(sm),dsubvertex(S))) AND
             atmost1?(intersection(DeepH(sm),dsubvertex(S))) AND
             atmost1?(intersection(ShallowH(sm),dsubvertex(S))) AND
             (S /= R ⇔
             intersection(dsubvertex(S),dsubvertex(R)) = emptyset) AND
             (isConcurrent(S) ⇒ every(compositeState?,dsubstate(S)) AND
             atleast2?(dsubstate(S))))
```

In other words, a composite state can contain at most one the following kinds of vertices: initial vertex, deep history vertex and shallow history vertex. A state vertex can be part of at most one composite state. A concurrent composite state can only have composite states as direct substates, and the number of these (direct) substates should be at least two.

The second rule labelled *wf2*, states that the top state of a state machine is always a composite state, it cannot have any containing states and cannot be the source of a transition. A state machine is aggregated within either a classifier or a behavioral feature.

```
wf2: THEOREM
    compositeState?(Root(sm)) AND
    outgoing(Root(sm)) = emptyset AND
    NOT (EXISTS (S: (State(sm))): S = container(Root(sm))) AND
    (context(sm) = behavioral OR context(sm) = classifier)
```

Rule *wf3* deals mainly with well-formedness of transitions and pseudostates. It states that an initial vertex can have at most one outgoing transition and no incoming transition. History vertices can have at most one outgoing transition. A join vertex must have at least two incoming transitions and exactly one outgoing transition. A fork vertex must have at least two outgoing transitions and exactly one incoming transition. Junction and choice vertices must have at least one incoming and one outgoing transition.

```
wf3: THEOREM
    (FORALL (S: (PseudoState(sm))):
             (member (S,Initial(sm)) ⇒
                     atmostst1?(outgoing(S)) AND
                     incoming(S) = emptyset)
             AND
             (member(S,union(Deeep(sm),ShallowH(sm))) ⇒
                     atmost1?(outgoing(S)))
             AND
             (member(S,Join(sm)) ⇒
                     atleast2?(incoming(S))
```

```
            AND singleton?(outgoing(S)))
    AND
    (member(S,Fork(sm)) ⇒
            singleton?(incoming(S)) AND
            atleast2?(outgoing(S)))
    AND
    (member(S,union(Junction(sm),Choice(sm))) ⇒
            incoming(S) /= emptyset AND
            outgoing(S) /= emptyset)
```

3.4 Formal Semantics of a Statechart Diagram

As mentioned above, we formally express a statechart as a transition system defined by a triple $(globalState, InitState, NEXT\_step)$. So the main goal of our formalization task consists of defining these three elements. We define a parametric theory called *FormalSemantics* which encompasses all the definitions involved. This theory receives two parameters: a value of type *StateMachine*, and a type representing the instance variables involved (corresponds to type V introduced earlier). We provide in this theory meaning functions for concepts such as event, state, action and condition, which are defined as predicates.

```
FormalSemantics [sm : StateMachine, V: TYPE] : THEORY
                BEGIN
                    meaningS: [State → PRED[[V, Time, Time]]]
                    meaningC: [Condition → PRED[V]]
                    meaningA: [Action → PRED[V]]
                    meaningE: [Condition → PRED[Time]]
                    EmptyE: Event
                    meaningE WITH[(EmptyE) :=
                      (LAMBDA(t: Time): true)]
                    nonemptyevent?(e: Event): bool = e/= emptyE
```

The meaning function of an action corresponds to its postcondition; the meaning function of an event corresponds to a predicate describing its occurence. For a condition, we use the condition predicate and for a state, a state predicate describing the activation status of the state.

We define a global state as a record type called *globalState* consisting of a state configuration and an event queue. The inital (global) state is defined as the combination of the initial configuration and a set of external events (i.e. generated by the environment). The initial configuration of a statechart corresponds to the set of default states involved.

```
globalState: TYPE = [ # conf: configuration, queue: finseq(Event) # ]
External: finseq(Event)
InitState: globalState =
          (# conf := C WHERE (C: configuration,
              FORALL(s: (C) | isSequential(s)): member(default(s),C)),
          queue := External #)
```

When an event is received, it is placed on the event queue of its target. An event is selected, dequeued and delivered for processing by an *event dispatcher mechanism.* The semantics of a state machine is based on the *run-to-completion* assumption. This means that events are dequeued and processed one at a time. The processing of an event by a state machine is called a *run-to-completion step.* A *run-to-completion step* brings the state machine from one stable configuration to another stable configuration. We describe *completion steps* execution by defining a binary relation denoted *NEXT_step.*

```
NEXT_step (s1: globalState,s2: globalState): bool =
                                        Eprocess(Edispatch(s1), s1, s2)
```

At the beginning of a step, an event is dispatched from the queue (operation *Edispatch*), then processed (operation *Eprocess*). Since the order of event dequeuing is not defined in the specification, we have decided to use the sequence order, *first arrived, first served.* The single restriction on this scheme concerns *deferred events.* An event instance that does not trigger any transition in the current state, will not be dispatched if it is defferred in that state. That gives the following definition for *Edispatch*:

```
Edispatch (s0: globalState): RECURSIVE Event =
      IF queue(s0) = empty_seq OR
        (EXISTS (i: nat): EmptyE = nth(queue(s0),i))
      THEN EmptyE
      ELSEIF
        (EXISTS (s: (conf(s0))): member(first(queue(s0)),defer(s)))
      THEN Edispatch(x)
        WHERE x: globalState, queue(x) := rest(queue(s0))
      ELSE first(queue(s0))
      ENDIF
MEASURE (LAMBDA (x: globalState): rest(queue(x)))
```

The processing of an event consists firstly of selecting a maximal set of non-conflicting transitions among the enabled one. The selected transitions are then fired, that consists for each of them of exiting from their source states, executing the associated actions and entering their target states.

```
Eprocess (e: Event, s0: globalState, s1: globalState): bool =
      (EXISTS (T: ({ A: set[Transition] | subset?(A, Transition(sm))
                     AND MaxEnabled(A,conf(s0),e) } )):
                         conf (s1) = union ({ s: (conf(s0)) |
                              intersection(outgoing(s),T)
                              = emptyset },
                              { s: (State(sm)) |
                              intersection(incoming(s),T)
                              /= emptyset } )
```

Due to possible conflicts among transitions, there are several maximal sets of transitions that can be fired. The actual set that is fired is chosen non-deterministically:

```
MaxEnabled( A: set[Transition] |
        subset? (A,Transition(sm)), c: configuration,
        e: Event(sm)): bool =
                FORALL (tr: (A)):
                        enabled(e,tr,c) AND
                        (FORALL (tr': (A)): NOT conflict(tr,tr'))
                        AND (FORALL (tr": (Transition(sm)) |
                        enabled(e,tr",c)
                        AND NOT member(tr",A)):
                              hasPriority(tr,tr") OR
                              samePriority(tr,tr"))
```

Two transitions conflict if there is at least one state that they both exit. In such situation, the selection of which transitions will fire is based on a priority scheme. By definition, a transition originating from a substate has a lower priority than any conflicting transition originating from any of its parent states. We give in the following the definitions corresponding to all these notions. We define before that the notion of *scope* or *least common ancestor* state of a transition. The scope of a transition is the lowest composite state that contains the explicit source states and explicit target states of the transition.

```
scope(tr: (Transition(sm))): State =
        s WHERE (s: (State(sm)),
                                isSequential(s) AND
                                member(source(tr),subvertex(s)) AND
                                member(target(tr),subvertex(s)) AND
                                (FORALL (s': (State(sm)) |
                                           isSequential(s')
                                           AND s /= s'):
                                (member(source(tr),subvertex(s')) AND
                                member (target(tr),subveretx(s'))
                                        ⇒
                                        member(s,substate(s'))))


conflict(tr: (Transition(sm)), tr': (Transition(sm))): bool =
            (tr /= tr') AND
            {s: (State(sm)) |
            (s= mainSource(tr)
            OR member(s,substate(mainSource(tr)))) AND
            (s= mainSource(tr')
            OR member(s,substate(mainSource(tr'))))
            } /= emptyset

hasPriority(tr1: (Transition(sm)), tr2: (Transition(sm))): bool =
                conflict(tr1,tr2) AND
                member(scope(tr2),substate(scope(tr1)))

samePriority( tr1: (Transition(sm)),tr2: (Transition(sm))): bool =
               (scope(t1) = scope(t2)) OR
               NOT (EXISTS (c: configuration):
                      member(source(tr1),c) AND member(source(tr2),c))
```

A transition is enabled if all its source states are in the active configuration, its trigger matches the event specified by the trigger and its guard condition is true.

```
enabled(e: Event, tr: Transition, c: configuration): bool =
           member(source(tr),c)
           AND match(e,trigger(tr)) AND
           meaningC(guard(tr))
```

The complete definition of theory *FormalSemantics* including the definitions of auxiliary functions and remaining semantic concepts is given in [Traoré 2000].

3.5 Properties of the Formal Semantics

Our formalization activity is based on the informal semantics of UML standard specification [OMG 1999], which serves as requirements document. The formal semantics provided as a PVS specification is validated by drawing conjectures from the requirements. We have identified and proved number of conjectures using the PVS prover. We present in the sequel two of these conjectures.

```
or_state: CONJECTURE
          (FORALL (s: (State(sm)) | isSequential(s))):
          meaningS(s) ⇔ singleton?({r: (dsubstate(s)) | meaningS(r) } )
and_state: CONJECTURE
           (FORALL (s: (State(sm)) | isConcurrent(s))):
           meaningS(s) ⇔ (FORALL (r: (dsubstate(s)): meaningS(r))
```

The first conjecture states that a sequential state is active if and only if one of its direct substate is active. The second conjecture states that a concurrent state is active if and only if all its direct substates are active (simultaneously).

4 Case Study

4.1 An Elevator System

Our case study concerns an elevator system that has been proposed and used in the literature as benchmark for several systems. The system consists of n elevator to be installed in a building with m floors. Each elevator has a set of buttons:

- One button for each floor that illuminates when pressed and make the elevator to visit the corresponding floor;
- An emergency button which when pressed generates a warning signal.

Each floor has two buttons (except the first and last floor), one to request an up-elevator and the other to request a down-elevator. When an elevator has no request, it should remain at its final destination, and wait for further requests. All requests for elevator from floors must be serviced eventually, with all floors given equal priority. All requests for floors within elevators must be serviced eventually, with floors being serviced sequentially in the direction of travel.

4.2 UML Specification

We define in the UML class diagram, a control class representing the elevator object. We describe the dynamic behavior of the *elevator* class by providing the statechart diagram depicted by figure 1. Initially the Elevator is considered to be in service (state *InService*). The elevator may be put out of service (state *OutService*) if a *cancel* event is generated. In this case the elevator may come back to the working state if an *ok* event occurs.

There are two major concerns when the elevator is in the working state: the update of the list of floor requested and the servicing of the individual requests. These two concerns are addressed by dividing state *InService* into two concurrent regions: *Dynamic* and *FloorRequested*. A request for a floor is serviced by updating the list and by moving at the same time the elevator in the appropriate direction. For instance, figure 2C depicts a sequence of stable configurations of the systems when a request is made. The request is made when the elevator is standing (state *Standing*) and the floor list is empty (state *ListEmpty*). This brings the system in a new configuration in which the elevator is still standing, but the list is non empty (state *ListNEmpty*). Then after a time period *dts*, marked by the occurrence of event *tm(stop,dts)*, the system moves to another configuration in which the elevator is moving (state *Moving*).

The conditions related to the transitions involved in the diagram are defined under the form of predicates of the instance variables in figure 2A. The actual meaning associated to the elementary states involved in the diagram are also predicates on the instance variables. Some of these definitions are given in figure 2B. For instance, state *Standing* is characterized by an elevator speed which is permanently null (e.g. $speed = 0$).

4.3 Formal Analysis

We define a theory called *Analysis*, in which the formal analysis takes place. This theory imports theory *AbstractSyntax* and instantiates the remaining theories (i.e. *WellFormedness, FormalSemantics*). Before instantiating the theories, we need to define the value of the *StateMachine* data corresponding to the Elevator statechart diagram. We also need to provide an actual type for type parameter *V*. We define *V* as a record type whose fields correspond to the instance variables involved (i.e. attributes and relationships).

```
Analysis : THEORY
        BEGIN
         IMPORTING AbstractSyntax
        % Instance variables
         Direction: TYPE = { DOWN, UP }
         Status: TYPE = { On, Off }
         m: VAR nat % number of floors
         Floor:below[m]
         EMPTY: set[Floor] = emptyset
         V: TYPE = [ # speed: nat,
                       direction: Direction,
                       floorList: set[Floor],
                       status: Status # ]
         v: VAR V
```

We define in detail all the elements (e.g. events, conditions, states etc.) involved in the diagram and their related set. We give for instance in the following the definition of states *ListEmpty* and *ListNEmpty*, and *SmStates* the set of all the states involved in the elevator statechart diagram. All these definitions are in principle generated automatically.

```
ListEmpty: State
dsubvertex WITH [(ListEmpty) := emptyset]
defer WITH [(ListEmpty) := emptyset]
meaningS WITH [(ListEmpty) := (lambda(v): floorList(v) =EMPTY)]
ListNEmpty: State
dsubvertex WITH [(ListNEmpty) := emptyset]
defer WITH [(ListNEmpty) := emptyset]
meaningS WITH [(ListNEmpty) := (lambda(v): floorList(v) /= EMPTY)]
                                              ...
SmState: set[State] = {s: State | s= ListEmpty OR
                                  s=ListNEmpty OR s= UP OR
                                  s=DOWN OR s=Standing OR
                                  s=Moving OR s= OutOfService OR
                                  s=Dynamic OR s= FloorRequested OR
                                  s= InService OR s= Elevator}
```

Then, we define the *StateMachine* value and instantiate the generic theories remaining.

```
sm: StateMachine =
              (# State:= SmState
                 StubState:= SmStub
                 SynchState := SmSynch
                               ...
                 Root:= SmRoot
              #)
IMPORTING WellFormedness[sm], IMPORTING FormalSemantics[sm,V]
```

By invoking the prover, the well-formedness rules defined in theory *Well-Formedness* are checked. As mentioned above, that can be handled automatically because the domains involved are all finite. Then, we define the properties of the system (within the same theory) and invoke the PVS model-checker. Model-checking in PVS requires the instantiation of theory *state* (see figure 4), which is already defined in the PVS prelude. We use as actual parameter for that state, the type *globalState*. We give also an initialization predicate *Init*, which is a state predicate defining the initial global state.

```
IMPORTING state[globalState]
s: VAR state
Init(s): bool = (s = InitState)
```

One of the most important properties for the elevator system is a liveness property stating that all requests will eventually be served. That corresponds to a state configuration in which the list of floor requested is emptied and the elevator is standing; we describe that by an assertion called *service*. The property itself is defined as a lemma stating that from an initial state, it is always possible to reach a state satisfying assertion *service*.

service(s): bool = member(ListEmpty,conf(s)) AND
member(Standing,conf(s))

live: LEMMA Init(s) ⇒
AF*(NEXT_step,service)(s)*

This property is automatically proved by using the *model-check* command of the PVS toolkit.

5 Concluding Remarks

In this paper we defined a formal semantics for a large subset of UML statecharts that serve for model-checking using the PVS toolkit. The advantage of model-checking over theorem proving is that model-checking is largely automatic. Moreover, model-checking technology has reached where it is possible to handle quite large state spaces. The size of the state space corresponding to the formal model proposed in this paper is function of the size of the fields of the abstract data corresponding to the statechart diagram. The elevator case study yields a state space with about 4.10^7 possible states. The formal model proposed can also serve for proof-checking. Actually, model-checking and theorem proving are both complementary techniques. Model-checking can be used to find counterexamples before starting some costly and time-consuming proof-checking.

We are currently developing an environment called *PrUDE* that automates the key steps of our formalization approach. PrUDE is a software verification and validation environment providing support for model execution, model-checking and proof-checking. Model-checking and proof-checking are based on the PVS toolkit. The automation of proof-checking will be enhanced by selecting and customizing suitable proof strategies provided by the PVS toolkit. The interface of PrUDE with UML is based on XMI, which provides an explicit interchange format for UML based tools. Since all UML tools export the UML model in XMI format, the platform will be independent of any tool vendors. It will then be possible to adapt it easily to existing software development environment. The main strength of PrUDE will rely on the fact that a user will have to deal only with graphical notations which are friendly, easy to learn and to use. All the formal "stuff" (related to PVS) will be processed at the back end. Another characteristic of PrUDE is the promotion of reusability by providing a library of formal models corresponding to existing frameworks and patterns.

Future work will deal with issues specific to object orientation such as inheritance, sub-behavior etc. We will mainly focus our work on statechart refinement policies (i.e. subtyping, behavioral compatibility and implementation reuse) which are still not clearly defined in the standard semantics document. We will also provide a detailed semantic for specific features of statecharts such as actions which have been defined in this work as uninterpreted types. Actually, there are no defined semantics for actions in the current UML specification. An accepted standard may be provided by 2000, as a result of the request for proposal (RFP) made by the OMG in November 1998 for a definition of the semantics of actions. Most works available in the litterature are dealing with the formalization of classical statechart. For instance, in [Hooman *et al.* 1992], a compositional semantics is provided for Harel Statecharts. Another approach proposed in [Chan *et al.* 1998] is based on a variant of Harel Statecharts, called Requirements State Machine Language (RSML) [Leveson *et al.* 1994] and targets symbolic model-checking. The behavior of the system is first represented in RSML, and then translated into input to the Symbolic Model Verifier (SMV) [McMillan 1993]. Closer to our work is the approach followed by Latella *et al.* in [Latella *et al.* 1999] where an operational semantics is provided for UML statechart diagrams by mapping them to an intermediate format of extended hierarchical automata. They also aim at model-checking as main verification technique, more specifically the model checking tool SPIN [Holzmann 1997].

References

[Abadi 1994] M. Abadi, L. Lamport, "An old-fashioned recipe for real-time", ACM Transactions on Programming Languages and Systems, 16:1543-1571, 1994.

[Chan *et al.* 1998] W. Chan, R.J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, J. D. Reese, "Model-Checking Large Software Specifications", IEEE Trans. Soft. Eng., Vol. 24, No. 7, July 1998.

[Dahl 1998] O.J. Dahl, O. Owe, "Formal Methods and the RM-ODP", Technical Report 261, IFI, University of Oslo, May, 1998.

[Demissie *et al.* 1999] D.B. Aredo, I. Traoré, K. Stølen, "An Outline of PVS Semantics for UML Class Diagrams", Research Report No. 272, University of Oslo, Norway. NWPT'99, Nordic Workshop on Programming Theory, Oct. 6-8, Uppsala, Swedden.

[Evans *et al.* 1999] A. Evans, S. Cook, S. Mellor, J. Warmer, A. Wills, "Advanced Methods and Tools for a Precise UML", 2nd International Conference on the Unified Modeling Language. Editors: B.Rumpe and R.B.France, Colorado, LNCS 1723, 1999.

[Evans 1998] A. Evans, "UML class diagrams - filling the semantic gap (draft)", Technical Report, York University, 1998.

[Hayes 1991] F. Hayes, D. Coleman, "Coherent Models for Object-Oriented Analysis". In OOPSLA conference proc., Phoenix, AZ, October 1991. Communications of the ACM.

[Holzmann 1997] G. Holzmann, "The Model checker SPIN", IEEE Trans. on Soft. Eng., 23(5):279-295, 1997.

[Hooman *et al.* 1992] J. Hooman, S. Ramesh and W.P. de Roever, "A compositional axiomatization of Statecharts, journal Theoretical Computer Science, 1992, vol. 101, pp. 289-335.

[ISO-lotos 1988] "LOTOS-A formal Description Technique Based on the Temporal Ordering of Observational Behavior", International organisation for standard, information processing Systems, open Systems Interconnection, geneva, Switzerland,Sept 1988. ISO Standard 8807.

[ISO-estelle 1989] ISO-IEC 9074: " ESTELLE - A Formal Description Technique based on an Extended State Transition Model", Geneva, Switzerland, 1989.

[ISO-sdl 1993] ITU Recommendation Z.100-CCITT, "Specification and Description Language (SDL)", 1993.

[ISO-rmodp] ISO-IEC JTC1/SC21/WG7, "The Reference Model of Open Distributed Processing", http://www-cs.open.ac.uk/ m_newton/odissey/RMODP.html

[Kneuper 1997] R. Kneuper, " Limits of Formal Methods", Formal Aspects of Computing (1997) 9: 379-394.

[Latella *et al.* 1999] D. Latella, I. Majzik, M. Massink, "Towards a Formal Operational Semanics of UML Statechart Diagrams", FMOODS'99, February 1999, Florence, Italy.

[Leveson *et al.* 1994] N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, J.D. Reese, "Requirements Specification for Process Control Systems", IEEE Trans. Soft. Eng., Vol. 20, no. 9, Sept., 1994.

[McMillan 1993] K.L. Mc Millan, "Symbolic Model Checking", Kluwer, 1993.

[Moreira 1994] A. Moreira, R. Clark, "Combining Object-oriented Analysis and Formal Description Techniques", In ECOOP'94, volume 821 of LNCS, Bologna, Italy, July 1994. Springer-Verlag.

[OMG 1999] "OMG UML Specification v.1.3", (OMG document ad/99-06-08), http://uml.shl.com/artifacts.htm

[Owre *et al.* 1995] S. Owre, J. Rushby, N. Shankar, F.V. Henke, "Formal Verification for Fault-tolerant Architectures: Prolegomena to the design of PVS", IEEE tran. on Soft. Eng., 21(2):107-125, February 1995.

[Shankar 1997] N. Shankar, "Machine-Assisted Verification Using Theorem Proving and Model Checking", Mathematical programming Methodology, (ed.) M. Broy, Springer-Verlag, 1997.

[Spivey 1989] J.M. Spivey, "The Z Notation: A Reference Manual", Prentice-Hall International, 1989.

[Traoré 2000] I. Traoré, "Making the UML More Precise: A Formal Framework for UML Statechart", Technical Report No. ECE 00-4, Department of Electrical and Computer Engineering, University of Victoria, October 2000.