Semantics of UML Statecharts in PVS

Demissie B. Aredo
Norwegian Computing Center
P. O. Box 114 Blindern, N-0314 OSLO, Norway.
E-mail: aredo@nr.no

ABSTRACT

In this paper, we propose formal semantic definition for UML *statecharts* in the PVS specification language. Based on the semantic definitions, we develop a general framework for translating UML statechart diagrams into PVS specifications, and show how the resulting specification can be model-checked by using the PVS toolkits. This work is a part of a long-term vision to explore how the PVS formalism can be used to underpin practical tools for checking correctness of UML models, and it contributes to the ongoing effort on providing precise semantic definitions for UML notations with the aim of clarifying the language as well as supporting the development of semantically based tools.

Keywords: Formal Semantics, UML, PVS, Method Integration, Statecharts

1 INTRODUCTION

The Unified Modeling Language (UML) [13] is an industrial standard for object-oriented modeling languages that was standardized by the Object Management Group (OMG). It is a collection of several description techniques which are suitable for modeling different aspects of software systems. Compared to other object-oriented modeling languages in software engineering, UML is more precisely defined and contains a great deal of formal specification notations, e.g. the use of Object Constraint Language (OCL) [17] for specifying constraint. However, semantic definitions for UML notations are not precise *enough* to support rigorous reasoning - a limitation that hampers its application to rigorous system development.

In the sequel, we propose formal semantics for the UML statecharts. Our aim is to achieve two goals. Firstly, we provide semantic model for basic modeling elements of UML statecharts using the PVS specification language [14]. This consists of formal representation of the abstract syntax and the well-formedness rules, and model-checking the resulting specification. Secondly, we propose a general scheme for translating UML statecharts into PVS specifications. This results in semantic models that are amenable to rigorous analysis. Using PVS tools such as the theorem-prover and model-checker, we rigorous reason about the resulting semantics models.

Several works have been undertaken to provide mathematical basis to the concepts underlying object-oriented (OO) models using different approaches and semantic foundations. In general, formalization approaches can be categorized into three: [5]: *supplemental*, *OO-extension* and *method-integration.* In the *supplemental* approach informal modeling notations are replaced by more formal constructs. The work of Moreira *et. al.* [12] is based on this approach and involves the LOTOS and the syntropy notations. The *OO-extension* approach extends existing formal methods by OO features thus making them more compatible with the concepts of object-orientation. For example, VDM++, Z++, and Object-Z are based on this approach. Even though a rich body of formal notation results from supplemental and extension approaches, the resulting semantic domain is more complex and suffers from lack of tool support [1],[3]. Moreover, users have to deal directly with a certain amount of formal artifacts. This is one of the major barriers for whole-scale utilization of formal methods due to their esoteric nature.

The *method-integration* [15] approaches makes OO notations more precise and amenable to rigorous analysis by integrating them with suitable formalism(s) [4]. It is a more workable and commonly used approach to formalization of OO modeling notations. The OO notation and a carefully chosen formalism, and their respective CASE tools are integrated allowing developers to manipulate the graphical models they have created without having an in-depth knowledge about the formal specifications that are processed at the back-end [3]. Our work is based on the method-integration approach and provides semantic definitions for UML statecharts using the specification language of PVS as underlying semantic foundation.

The rest of the paper is organized as follows: In Section **2**, a brief overview of the PVS specification language is presented with emphasis put on concepts and notations that will be encountered in later sections. In Section **3**, mail concepts of UML *statecharts* are discussed. In Section **4**, semantic definitions for the basic concepts of UML statecharts are proposed. Finally, in Section **5**, we draw some conclusions and discuss future works.

2 THE PVS ENVIRONMENT

PVS [14] is a formalism for design and analysis of system specifications. It consists of a highly expressive *specification language* tightly integrated with a *type-checker* a *theorem-prover*, and other tools. A strength of PVS is its capacity to exploit the synergy between the specification language and its tools, e.g. the type-checker uses the theorem-prover. The theorem-prover allows construction of proofs interactively and rerun them automatically after minor changes.

The PVS specification language (PVS-SL) provides a very general semantic foundation based on the classical higher-order logic. Its type system consists of basic types such as *boolean, integer, real,* and constructors for *set, tuple, record,* and *function* types. A record type consists of a finite set of fields `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. Given a record `r:R`, a function application-like term $a_i(r)$, is used to access the i^{th} field of `r`. Tuples have similar structures except that the order of fields is significant in tuples. A function type is specified as `F:TYPE= [`$D \rightarrow R$`]` where D and R are type expressions denoting domain and range of the functions. For a given type `T`, the type of sets of elements of `T` is specified using one of the constructs `pred[T]` or `setof[T]`, each of which is a shorthand for the predicate `[T` $\rightarrow$ `bool]`. For a given set `s:setof[T]` and `t:T`, membership of `t` in `s` is determined by the truth value of `member(t,s)`, or `s(t)`.

The type system of the PVS-SL has been augmented by *predicate subtyping* and *dependent typing* mechanisms and supports a richer type system than the classical higher-order logic. Subtyping makes type-checking more powerful and allows stronger checks for consistency and invariance in a uniform manner [2]. However, it renders type checking undecidable as a result of that the type-checker generates proof obligations called *Type Correctness Conditions* (TCC's). A great deal of TCC's are discharged automatically, whereas more involved ones require interactive use of the theorem-prover. Predicate subtypes can be specified in two different ways. Given a type `T` and a predicate `p` on elements of `T`, a predicate subtype of `T` with respect to `p`, can be specified as either `S:TYPE= {t:T | p(t)}` or `S:TYPE= (p)`. When the expression of the predicate is not explicitly given, we can specify `S` as uninterpreted subtype of `T`, symbolically `S: TYPEFROM T`.

The PVS prover provides primitives to perform inductive reasoning, rewriting, and model checking. These features simplify the proof process as mechanical aspects can easily be automated. quite easily [8]. Specifications in PVS are organized into hierarchies of *theories*. A theory may contain type, variable, and constant declarations, definitions, axioms, and theorems. Modularity and reusability are captured by parameterized theories that specify generic elements that are instantiated by *theory abbreviation* construct. Predicates, usually known as *assumptions*, are used to constrain the parameters of a generic theory. PVS-SL includes a library of an extensive set of built-in constructs known as *preludes*, that provides several useful definitions and lemmas.

A detailed presentation of the PVS environment is beyond the scope of this paper. For a more complete and detailed discussion, interested reader may refer to [14].

3 UML STATECHARTS

UML statecharts [13] are primary modeling elements for construction of executable models that capture complex dynamic behavior of reactive systems. A statechart describes an abstract machine that defines a set of existence conditions, called *states*, a set of behaviors or *actions* that can be performed in each of those states, and a set of *events* that may cause state *transitions* according to a set of well-defined rules.

A statechart describes a model element in isolation in terms of its interaction with the rest of the world by responding to certain events. A response of an object to an event, and the action that may ensue as a result depend on the current state of the object and the event that occurs. This may possibly result in performance of an action and a transition into another state. An event may cause a firing of a transition, and execution of a sequence of actions associated with the transition. When the object modelled by the state machine is in a given state, it reacts only to certain events by performing the corresponding actions, and may transform into a subset of the set of states.

UML statecharts are object-oriented variants of the classical statecharts first conceived by Harel [7]. The main difference between the UML statecharts and the classical ones is that the former specifies behavior of *types* whereas the latter specifies behavior of *processes*. In fact, the notion of process is not supported in the UML. The classical statecharts assume *zero-time* transition, whereas a transition may take some time in the UML statecharts; events are not broadcasted in UML, but they may be sent to a set of objects. For a detailed comparison between UML statecharts and the classical statecharts, interested readers may refer to chapter 2 page 157 in the standard document of UML version 1.3 [13].

In the context of object-oriented modeling techniques, elements that can have dynamic states are objects. Objects have both structural and behavioral properties. Static structural aspects of objects are described by UML class diagrams, whereas behavioral aspects can be captured by statechart and interaction diagrams. A state machine is associated to a specific modeling element, usually an object or an interaction, and specifies complete dynamic behavior of the modeling element by describing its reaction to events. The associated modeling element determines the context of a state machine. A typical instance is the use of state machines to model the behavior of reactive objects by describing their complete life cycle.

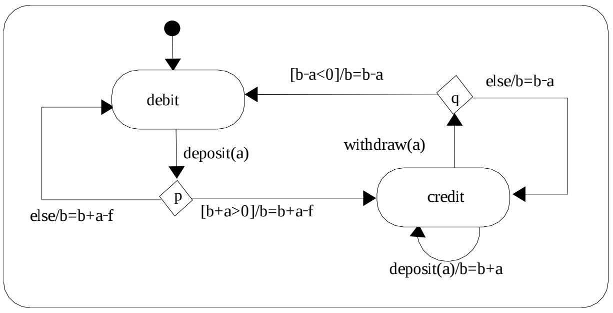

Fig. 1. UML statechart for an Account Class

An example of a UML statechart diagram shown in Figure 1 specifies a complete life cycle of an account object. An account can be either in the `debit` or the `credit` state depending on the value of its attribute balance *b*. The banking system allows customers to withdraw a given amount of fund in debt, subject to fixed fee *f*, hence the introduction of the `debit` state of the account. When an object is in the *debit* state, `deposit(a)` is the only operation allowed. At junction `p`, a guard condition `[a+b>0]` is evaluated to check the amount against the balance `b`. Note that the balance `b` is less than zero when the account is in the `debit` state, and hence the deposited amount must be compared to `-b`. If the guard condition `[a+b>0]` is true, the account is transformed into the *credit* state, otherwise it remains in the *debit* state. In any case, the balance is updated by computing `b:=b+a-f`, where `f` is some constant fee charged when the account is in `debit` state. When an account object is in the *credit* state, the `deposit(a)` event increases its balance by, and leaves its state unaltered. An occurrence of a `withdraw(a)` event when the account is in *credit* state, may transform it into the *debit* state or leave it in the same state depending on the truth value of the guard condition `[b-a<0]` at junction `q`. In either case, the balance is updated with the result of `b:=b-a`.

4 SEMANTICS OF UML STATECHARTS

In this section, we provide semantic definitions for UML statecharts by transforming them into appropriate entities in the PVS specification language. We encode the abstract syntax of UML statecharts, and associated well-formedness requirements. Note that the PVS-SL is used as underlying semantic foundation and not as a description language and hence users are not expected to have an in-depth knowledge about neither the PVS-SL nor its proof system. We define semantic models for statecharts using bottom-up approach, i.e. starting with semantic definitions of basic model elements such as *states, transitions, events* and *actions* we provide semantic definition for statecharts as an appropriate composition of semantic definitions of its components. We treat the informal semantics descriptions provided in UML version 1.3 standard document [13] as a requirement specification on which the formal semantic models will be based. Some constraints on UML models may involve dynamic information, e.g. the number of objects created could only be available during run time.

We specify a parameterized theory that defines a predicate on sets of elements of a type given as parameter of the theory. The predicate `optional?()` filters the empty set and singleton sets of elements of the type.

```
optional[T : TYPE] : THEORY
 BEGIN
  x, y :  VAR T; s :  VAR set[T]
  singleton?(s):bool= EXISTS(x:(s)):
                           FORALL (y:(s)):  y=x
  optional?(s):bool= (empty?(s) OR singleton?(s))
 END optional
```

Given a type `T` and a set `s` of elements of `T`, `(s)` denotes a subtype of `T` containing exactly the elements of `s`. For every type (class in the UML vocabulary) involved in optional multiplicity, a new theory is instantiated from the generic theory *optional* with the type as a parameter using the PVS construct known as *theory abbreviation.* For instance, for the type `T`, a theory `optional[T]` is defined as an instance of theory `optional`. The expression `optionalT.optional?` provides access to the predicate `optional?`.

```
optionalS : THEORY = optional[T]
s :  (optionalT.optional?)
```

4.1 Abstract Syntax of UML Statecharts

We start by representation of the notions of *model element, action, signal*, and *operation* as uninterpreted types in the PVS specification language. The `ModelElement` is a root class from which every class in UML metamodel inherits. The details of these model elements are intentionally avoided since such details are irrelevant at the level of abstraction we are working.

```
ModelElement :  TYPE+
Action,Signal,Operation:  TYPE FROM ModelElement
```

Next, we discuss notions of *states, transitions* and *statecharts*, and formally represent them.

States: A state is a specification of a snapshot of values of program variables or behavior of an object that satisfies some, usually implicit, invariant conditions. Objects of a given class that are in the same state have the same qualitative responses to an occurrence of the same event. That is, they react to events in the same way, and execute the same sequence of actions, and may undergo the same set of transitions (apart from non-determinism).

A state vertex is an abstraction of a node in a statechart diagram. In the UML meta-model, state is a direct subclass of the class `ModelElement` and hence we represent it as a subtype of the type `ModelElement`. In general, a state vertex can be a source and target of any number of transitions. In the record type *State*, the field `asModelElement` captures properties inherited from the superclass `ModelElement`.

```
StateVertex :  TYPE FROM ModelElement
```

The class `StateVertex` can be specialized into the following four kinds of states: `State, PseudoState`, `StubState`, and `SynchState`. A synchronous state is used to synchronize concurrent regions of a state machine. Pseudo states are vertices in the state machine that are used to connect multiple transitions into a transition path. A stub state appears within a submachine to refer to the actual subvertex contained within the referenced state machine. A state may have an **entry** action - the first action that takes place when the state is entered, a set of **internal** transitions and associated actions, and an **exit** action - the last action that takes place when the state is exited.

Usually, an event that does not enable a transition is discarded. However, it is sometimes useful to keep this event waiting until the next state. A set of events to which a state machine does not react while it is in a given state is described as a set of "deferable" events - the field `deferable` captures a set of such events. Note that we declare variables only once and use them in the later sections.

```
T: TYPE; x, y:  VAR T; s :  VAR set[T]
optionalAction :  THEORY = optional[Action]

State:  TYPE =
        [# asStateVertex:  StateVertex,
        entry:  (optionalAction.optional?)),
        doActivity:  (optionalAction.optional?)),
        exit:  (optionalAction.optional?)),
        deferable:  setof[Event]#]

PseudoStateKind:TYPE= {initial,deepHist,join,
                shallowHist,fork,junction,choice}
PseudoState:TYPE=[# asStateVertex:  StateVertex,
           pseudoKind:  PseudoStateKind #]
StubState:TYPE= [# asStateVertex:  StateVertex,
                refState:  String #]
SynchState:TYPE= [# asStateVertex:  StateVertex,
                bound:  nat #]
```

The class `State` is further specialized into `SimpleState`, `CompositeState`, and `FinalState` which we represent as subtypes. A composite state can be concurrent or sequential.

```
v :  VAR StateVertex
SimpleState :  TYPE FROM State
```

```
FinalState : TYPE = {v | outgoing(v) = ∅}
CompositeState : TYPE =
         [# asState : State,
         isConcurrent : bool,
         dsubstate : fin_set[StateVertex] #]
container : [StateVertex → CompositeState]
```

The `container` function returns the smallest composite state (if any) that contains a state vertex. The field `dsubstate` captures the set of direct sub-states of a state. It is used to define the function `subvertex()` which returns the set of all sub-states of a given composite state. The `subvertexInc()` returns the set of sub-states of a state including the state itself. When applied to the `top` state of a state machine, `subvertexInc()` returns the set of all state vertices in the state machine by recursive application of `dsubstate()` to the vertices.

```
contains(v,cs): bool = CompositeState(cs) ∧
                        member(v, dsubstate(cs))
subvertex(cs): RECURSIVE setof[StateVertex]=
               union(dsubstate(cs),
                      ⋃_{v∈dsubstate(cs)} subvertex(v))
      MEASURE (LAMBDA cs: dsubstate(cs) ≠ ∅)

subvertexInc(cs): setof[StateVertex] =
                  union({cs},subvertex(cs))
```

If an event is deferred in a given composite state, then it is deferred in any substate of that state. We add the axiom *deferax* given below to captures this notion.

```
v,v′: VAR StateVertex; cs: VAR CompositeState
deferax: AXIOM (v∈subvertexInc(cs)) ⇒
               (deferable(cs) ⊆ deferable(v))
```

Transitions: A transition in UML statecharts models a change in object behavior from one state to another state (not necessarily distinct) as a result of a response to a reception of an event. The set of transitions specifies a reaction of an object to events, or the action carried out by its methods in response to occurrence of the event. In other words, an object in a given state, called the *source* of transition, evolves into another state, called *target* state, when a specific event occurs and a *guard* condition is satisfied, and perform a sequence of actions.

A transition in a statechart may be labelled by a string of the form `e[c]/sa`, which means that the occurrence of event `e`, when the guard condition `c` is true, triggers the firing of the transition, as a result of which the object performs sequence of actions `sa`. The UML standard [13] also allows triggerless transitions, called *completion transitions*. They have implicit triggers, i.e. *completion event*, which are generated when all transitions, entry actions and activities in the currently active state are completed.

To define semantics of a transition, we need the types `Event`, `Action`, and `Guard`, and instances of the theory `optional` instantiated with these types. Then, the notion of transition is captured by a record type with appropriate set of fields.

```
Event : TYPE FROM ModelElement
Guard : TYPE = [# asModelElement: ModelElement,
                  expression: BoolExpression #]

optionalEvent : THEORY = optional[Event]
optionalGuard : THEORY = optional[Guard]
optionalAction : THEORY = optional[Action]

Transition: TYPE =
    [# asModelElement : ModelElement,
       source  : StateVertex,
       trigger : (optionalEvent.optional?),
       guard   : (optionalGuard.optional?),
       effect  : (optionalAction.optional?),
       target  : StateVertex #]
```

We define some operations that specify associations between states and transitions. The functions `incoming()` and `outgoing()` defined on `StateVertex` return, respectively, the set of transitions entering and leaving the vertex. A transition connects exactly one source state and one target state, which are retrieved by applying the accessor functions `source` and `target` respectively, to the transition record.

```
incoming : [StateVertex → setof[Transition]]
outgoing : [StateVertex → setof[Transition]]
```

State Machines: A state machine can be described completely by a *top* state, i.e. a composite state at the root of the state containment hierarchy, and a set of transitions. Given the top state of a state machine and the set of its transitions, all the remaining states can be retrieved by traversing the state containment hierarchy starting at the top state. Application of the `subvertexIncl()` function described above to the top state of a state machine returns the set of all state vertices in the state machine.

Semantics of a state machine is defined as a record type whose set of fields contain the *top* state vertex, and the set of transitions. Symbolically,

```
StateMachine: TYPE =
              [# asaModelElement: ModelElement,
                 top: StateVertex,
                 transitions: setof[Transition]
                 context: ModelElement] #]
context : [StateMachine → Context]
```

The function `context()` determines the model element whose behavior is captured by the state machine. A model element can be described by several state machines, but a given state machine describes at most one model element. The specification of function `context()` ensures that this requirement is fulfilled.

The `SubmachineState` defined below is a syntactical convenience that facilitates modularity and reuse, and is semantically equivalent to a composite state. It is a placeholder for a state machine that is referenced by another state machine. The `submachine()` function defined below determines the state machine for which a submachine state stands in a given composite state. The `stateMachine()` function returns the state machine to which a transition belongs.

```
SubmachineState : TYPE FROM CompositeState
submachine: [SubmachineState,CompositeState→
                                  StateMachine]
stateMachine : [Transition → StateMachine]
```

4.2 Well-formedness Requirements

In this section we formalize well-formedness requirements (WFRs) on some of the modeling elements described above. The well-formedness rules can be defined in the same theory as the model elements they constrain or in a separate theory and imported. We follow the latter option since this approach matches the informal descriptions given in the standard document of UML v1.3 [13]. The WFRs are labelled with the labels in the UML standard document [13] suffixed with the initial letter of the model element they constrain. For instance, *ruleCS1* corresponds to the first well-formedness rule for composite state.

```
s :  VAR State;        c1 :  VAR CompositeState
v :  VAR StateVertex;  m  :  VAR StateMachine
ps:  VAR PseudoState;  t  :  VAR Transition
```

WFRs of Composite States: The following WFRs apply to *CompositeState*. A composite state can contain at most one vertex of each of the pseudostate of *initial, deepHist, and shallowHist* kind.

```
ruleCS1(cs):  bool =
 optional?({ps|ps∈subvertex(cs) ∧
                pseudoKind(ps) = initial})
 ∧ optional?({ps|ps∈subvertex(cs)∧(ps)=deepHist})
 ∧ optional?({ps|ps∈subvertex(cs)∧
                PseudoKind(ps)=shallowHist})
```

A concurrent composite state must have at least two direct subvertices each of which is a composite state.

```
ruleCS2(cs):  bool = isConcurrent(cs) ⇒
        ((‖subvertex(cs)‖ ≥ 2) ∧
                (subvertex(cs) ⊆ CompositeState))
```

where ‖.‖ is a function that returns the cardinality of a set. A given state vertex can be a part of at most one composite state.

```
ruleCS3(v):  bool =
   (v∈substate(cs) ∧ v∈substate(c1)) ⇒ cs=c1
```

WFRs of Transitions: A fork segment should not have guards or triggers:

```
ruleT1(t):  bool= (PseudoState(source(t))
                  ∧ PseudoKind(source(t))=fork)⇒
                  (guard(t)=∅ ∧ trigger(t)=∅)
```

A join segment should not have guards or triggers.

```
ruleT2(t):  bool= (PseudoState(target(t))
                  ∧ pseudoKind(target(t))=join)⇒
                  (guard(t)=∅ ∧ trigger(t)=∅)
```

A fork segment should always target a state:

```
ruleT3(t):  bool= (stateMachine(t)≠∅
                  ∧ PseudoState(source(t))
                  ∧ PseudoKind(source(t))=fork)⇒
                          State(target(t))
```

A join segment should always originate from a state:

```
ruleT4(t):  bool= ((stateMachine(t) ≠ ∅ ∧
                   PseudoState(target(t)) ∧
                   pseudoKind(target(t)) = join)
                       ⇒ State(source(t))
```

Transitions outgoing from a pseudostates may not have a trigger:

```
ruleT5(t):  bool = PseudoState(source(t))⇒
                                  trigger(t) = ∅
```

Join segments should originate from orthogonal states:

```
ruleT6(t):  bool= (PseudoState(target(t)) ∧
                   pseudoKind(target(t))=join)
                ⇒ isConcurrent(container(source(t)))
```

Fork segments should target orthogonal states:

```
ruleT7(t):  bool= (PseudoState(source(t)) ∧
                   pseudoKind(source(t))=fork)
                   ⇒ isConcurrent(target(t))
```

An initial transition at the topmost level may have a trigger with the stereotype "create". An initial transition of a StateMachine modeling a behavioral feature has a CallEvent trigger associated with that BehavioralFeature. Apart from these cases, an initial transition never has a trigger:

```
CallEvent :  TYPEFROM Event
stereotype :  [ModelElement → ModelElement]
ruleT8(t):  bool= (PseudoState(source(t)) ∧
                   kind(source(t))=initial)
⇒(trigger(t) = ∅
   ∨ (container(source(t))=top(stateMachine(t))∧
      name(stereotype(trigger(t)))="create")
   ∨ (BehavioralFeature(context(stateMachine(t)))∧
   CallEvent(trigger(t))∧
   operation(trigger(t))=context(stateMachine(t))))
```

WFRs of State Machines: A state machine is aggregated either within a classifier or a behavioral feature. The context of a state machine should be an object or a behavior as specified by the well-formedness requirement *ruleSM*1 given below.

```
ruleSM1(m):  bool= Classifier(context(m)) ∨
                   BehavioralFeature(context(m))
```

The following three expressions specify the facts that the top state of a state machine is always a composite state, the top state doesnot have a container state, and it cannot be the source of a transition.

```
ruleSM2(m):  bool= CompositeState(top(m))
ruleSM3(m):  bool= container(top(m)) = ∅
ruleSM4(m):  bool= outgoing(top(m)) = ∅
```

If a state machine describes a behavioral feature, it contains no trigger of type *CallEvent*, apart from the trigger on the initial transition.

```
ruleSM5(m):  bool = BehavioralFeature(context(m))
             ⇒ (∀ t:  t∈transitions(m) ∧
                 NOT (PseudoState(source(t)) ∧
                 pseudoKind(source(t)) = initial)
                 ⇒ trigger(t) = ∅)
```

4.3 Semantic Definitions

Once the abstract syntax of basic elements of UML state machines, and well-formedness requirements are precisely encoded in the PVS specification language, providing semantic definitions for more complex model elements is easier. Formalizing semantics concepts of UML state machines paves a way for specifying important properties exhibited by the system and for rigorous reasoning about their correctness.

In general, for a UML model `M`, whose abstract syntax is encoded in the PVS-SL as `SyntaxM` and its weel-formedness requirements as predicates `ruleM1, ..., ruleMk`, its semantics `SemM` is the predicate subtype of `SyntaxM` with respect to the conjunction of its well-formedness predicates. For instance, semantics of the state machine is defined as follows:

```
SemStateMachine :  TYPE=
        {m| ruleSM1(m) ∧ ...∧ ruleSM5(m)}
```

A state is said to be *active* when it is entered as a result of transition and becomes *inactive* when it is exited. A state can be thought of as a predicate on the set of program variables. The state is active when this predicate returns value `true`. For a composite state that is active, and non-concurrent, exactly *one* of its substates is active. If a composite state is active and concurrent, then *all* of its substates are active.

```
active:  [StateVertex → bool]
activeAx1:  AXIOM (active(c) ∧
                   NOT isConcurrent(c) ∧
                   v∈subvertex(c)) ⇒
             ||{v:(dsubstate(c))|active(v)}|| = 1

activeAx2:  AXIOM (active(c) ∧ isConcurrent(c) ∧
                   v ∈subvertex(c)) ⇒
       (FORALL (v:(dsubstate(c))) :  active(v))
```

If a give simple state is active, then every composite state containing the state, directly or transitively, is also active. Since some of the composite states may be concurrent, a current active state is represented by a tree of states, called *state configuration*, starting with the *top* most composite states down to individual simple states at the leaves.

```
configuration :  [StateMachine → setof[State]]
configuration(sm) = {s | s∈subvertex(top(sm)) ∧
                         active(s)}
```

More advanced semantic concepts such as conflicting transitions, firing priorities, etc. can similarly be formalized in terms of the basic concepts of UML statecharts defined above.

5 CONCLUSIONS

We have proposed semantic definitions for UML statecharts using the PVS specification language as underlying semantic foundation. The main objective of the work is to give a precise and equivocal description of the UML statecharts. Such a precise description is required as a reference model for implementing tools for code generation, simulation and verification of UML statecharts. The framework integrates a UML CASE tool and the PVS toolkit resulting in heterogeneous platform that combines the strengths of a semi-formal graphical modeling notation and a formal verification environment. Other benefits of transforming the UML statecharts into the PVS-SL include the ability to produce precise and analyzable specifications, and the availability of PVS toolkit that supports rigorous reasoning about the resulting semantic models.

Several semantics for statecharts have been proposed in the literature [7],[6],[9],[16]. Most of them are concerned with defining semantics of the classical Harel's statecharts [7]. For instance, Harel *et al.*[7],[6] present semantics of classical statecharts in the STATEMATE system. Mikk *et al.*[11] propose formal semantics of UML statecharts based on hierarchical automata. The representation in hierarchical automata is not suitable for tool development [10]. It does not directly support transition across compound states, and the hierarchical structure must be flattened before using it in a model checker. The work presented in the sequel is similar to the work presented in [16], yet this work is more detailed.

This work contributes to the ongoing effort to provide formal standard semantic definitions for UML notations, 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 CASE tools to analyze UML models.

Acknowledgements

I would like to thank Prof. Olaf Owe, Dr. Wenhui Zhang, and Dr. Issa Traore for their invaluable comments. This work was funded by the Research Council of Norway through the ADAPT-FT project.

REFERENCES

[1] J.-M. Bruel and Robert B. France. Transforming UML Models to Formal Specifications. In *the Proc. of the OOPSLA'98 Workshop on Formalizing UML. Why? How?*, Vancouver, Canada, October 1998.

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

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

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

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

[6] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.

[7] D. Harel, A. Penueli, J. P. Schmidt, and R. Sherman. On the Formal Semantics of Statecharts. In *the Proc. of the 2nd IEEE Symposium on Logic in Computer Science*, pages 54–64, New York, USA, 1987. IEEE Press.

[8] P. Krishnan. Consistency Checks for UML. In *the Proc. of the Asia Pacific Software Engineering Conference (APSEC 2000)*, pages 162–169, December 2000.

[9] D. Latella, I. Majzik, and M. Massink. Towards a Formal Operational Semantics of UML Statechart Diagrams. In *the Proc. of FMOODS'99, Florence, Italy.* Kluwer, February 15-18, 1999.

[10] J. Lilius and I. P. Paltor. The Semantics of UML State Machines. Technical Report No. 273, May 1999. Turku Centre for Computer Science, Finland.

[11] E. Mikk, Y. Lakhnech, and M. Siegel. Hierarchi-

cal Automata as Model for Statecharts. In K. Ueda R. K. Shyamasundar, editor, *the Proc. of Asian Computing Science Conference (ASIAN'97)*, volume 1345 of *LNCS*, pages 181–196. Springer Verlag, December 9-11 1997.

[12] A. Moreira and R. Clark. Combining Object-oriented Analysis and Formal Description Techniques. In *the Proc. of ECCOP'94, LNCS*, volume 821, Bologna, Italy, 1994. Springer-Verlag.

[13] The OMG. OMG Unified Modeling Language Specification, version 1.3, June 1999. OMG standard document.

[14] S. Owre, N. Shankar, J. Rushby, and D. W. Stringer-Calvert. PVS Language Reference, version 2.3, September 1999. Computer Science Laboratory.

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

[16] I. Traore. An Outline of PVS Semantics for UML Statecharts. *Jounal of Universal Computer Science*, 6(11):1088–1108, 2000.

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