The Formal Foundations in RSL for UML Statechart Diagrams∗

Sun Meng, Zhang Naixiao
LMAM, School of Mathematical Science
Peking University
Beijing, China, 100871
sunmeng@water.pku.edu.cn
znx@pku.edu.cn

Bernhard K. Aichernig
International Institute for Software Technology
United Nations University
Macau, P.O. Box 3058
bka@iist.unu.edu

Abstract

In this paper, we present a formalization for UML statechart diagrams in the RAISE specification language RSL. By such a formalization, we propose a general framework for integration of graphical UML statechart diagrams and formal RSL specifications, which forms the continuation of the previous work on formalization of UML class diagrams in RSL. This allows the definition of UML semantic interpretations that are precise and unambiguous, and also enhancing the readability, conciseness and abstraction of the resulting RSL specification. In a case study, we illustrate how the framework can be used to create formal specification for UML models and analyze the properties of the models.

Keywords: UML, statechart, RAISE, formalization

1 Introduction

Object-orientation has now become a popular approach in software industries [9, 10]. The Unified Modeling Language (UML) [11, 1], which is a graphical language for specifying, visualizing, constructing and documenting object-oriented (OO) systems, has become a de facto standard for OO modelling. The main advantage of UML is that it is a unification of different view models. Each model is intended as a complete description of the system from a particular perspective and describes a specific aspect of the system to be developed, such as the static structure of a class or a component of the system, the dynamic behavior of single objects in the system, and the communication and coordination between different objects in the system, etc. All these models together describe the system being designed.

∗This work is partially supported by the National Natural Science Foundation of China under Grant No. 60273001.

The graphical notations in UML are recognized as being intuitive and resulting from modelling experience. However, UML also suffers from the usual problem of semi-formal graphical languages: it lacks a precise semantics. This makes UML not amenable to formal reasoning techniques and constitutes a possible source of errors in development process. The definition of a formal semantics for UML has become an active research area recently. In general, The formalization works can be grouped into two major categories of approaches: One approach is to define a semantic domain and give a sound semantics in this domain to the graphical notations. Nevertheless, the plentiful graphical notations make the semantic domain complex, and always suffer from lack of analysis tool support [2]. This obstruction makes combining graphical notations with formal specification languages a more workable approach for formalizing the graphical modelling notations, and exploring the formal reasoning capabilities of formal methods.

Several studies have been undertaken to combine the semi-formal OO models and formal techniques. Especially, there has been much interest in integrating formal methods to UML to improve its shortfalls. Work has been done on the basis of well-established traditional formal methods such as Z [3] and Object-Z [12], and combining them with the notations in OO modelling [2, 13]. Works on integrating UML and algebraic specification techniques are provided in [4], [14] provides a translation from UML class diagrams to CCSL, which is a coalgebraic class specification language. However, most of these works only focus on formalization of UML class diagrams, and give up the advantage of unifying different models in one language.

In this paper, we present a formalization of the UML statechart diagrams via the RAISE Specification Language (RSL) [16, 17], which is a wide-spectrum language for specifying and designing software systems. The notion of statechart, first introduced by Harel [7], has been found versatile to be used in the context of OO system development. Having this formalization as basis, an integration of for-
nal and semi-formal methods can be achieved which will have both the understandability of the semi-formal UML notations and the unambiguity of RAISE. This work forms a continuation of the previous work on formalizing UML class diagrams in RSL [5]. The versatility and comprehensiveness of RSL makes it possible to integrate different UML view models into a unifying formal framework. By formalizing the elements of statecharts in RSL, a framework for transforming UML statechart diagrams to RSL specifications is given. Once a UML statechart diagram is translated into a RSL specification, the RSL tools also become available for rigorous analysis, reasoning of system properties and code generation from specifications.

The organization of this paper is as follows: The translation from UML static diagrams to RSL specifications is shown briefly in Section 2. Section 3 provides the formalization of statechart diagrams in RSL. The formal semantics for statecharts is discussed in Section 4. In Section 5, a case study is presented to show the application of the formalization framework. In Section 6, we draw the conclusion and discuss some future work issues. A complete set of well-formedness rules for statechart diagrams in RSL is given in Appendix.

2 Translating UML Static Models into RSL Specifications

In [5], a formalization of UML class diagrams in RSL has been given. UML class diagrams, augmented with additional well-formedness rules being written in Object Constraint Language (OCL), are used for the meta-model of UML representing the static properties of the basic elements of other UML models, including statechart diagrams.

On the other hand, UML class diagrams are used for describing the static properties of a system while statechart diagrams are used for the dynamic behavior aspects of a system. Since rigorous reasoning about the properties of a system may involve both static and dynamic aspects, analysis of one view model may require pieces of information provided by another, it is meaningful to relate the formal descriptions of both class diagrams and statechart diagrams. Once the two different notations are formalized in the same formalism, and the CASE tools are integrated, developers can directly manipulate the graphical artifacts without knowing the underlying formal stuff.

A class diagram consists of a set of classifiers, i.e. classes and interfaces, connected by their various static relationships. A class is a description of a set of objects or class instances with the same structural and behavioral features.

Each attribute has a name, an optional type, a multiplicity, a scope (classifier or instance), and a changeability (frozen, add only or changeable). An operation consists of a name, a list of formal parameters, where each parameter has a name and an optional type. Furthermore, an operation has an optional result-type, a scope and can be abstract. All this properties are specified formally in RSL.

Different classes in a class diagram are related by different kinds of relationships. Basically, they are classified into three types: associations, generalizations and dependencies. Instantiations are separated from general dependencies for its particular meaning.

Each relationship must be well formed. Therefore, the structure for each type of relationship together with the well formedness rules is defined as follows:

\[
\text{type} \quad \text{Rel} = \{ | r : \text{Rel1} \cdot \text{well_formed}(r) \},
\]

An association is a relationship among classes. Each association end holds information not only about the class but it has several adornments: an optional role name, a multiplicity, a navigability, a changeability, an aggregation kind, etc. Moreover, each association has an assigned name.

Generalizations, dependencies, as well as instantiations, are all binary relationships. In order to ensure that all the relationships in class diagrams are wellformed, a family of wellformedness rules for all the relationships specified in RSL is given in [5].

3 Translating UML Statecharts into RSL Specifications

In this section, we present the formal foundations for UML statechart diagrams by transforming them into RSL. We start by representation of the notions of the generic ModelElement as an uninterpreted type in RSL and put it in the scheme TYPE. This generic type gives us flexibility to parameterize different kinds of model elements.
Another basic notion is Action which is specified as follows:

```
context: TYPE
scheme ACTION =
class
  object
    T : TYPE
type
  Action,
    PName = Text,
    Parameter = PName × T.Type
value
  parameters : Action → Parameter
end
```

The syntax of UML statechart diagram is presented using RSL here, which is based on the abstract syntax for state machines in UML specification [11]. We give the syntax in a bottom-up fashion, starting by the states and transitions, which is different from the top-down style adopted in [5] for class diagrams. A statechart diagram is formed by states and transitions among them. In order to build a syntactically correct statechart diagram, a set of well-formedness rules must be satisfied on the states, transitions and the whole statechart diagram, which will be given later.

3.1 States

A state specifies a condition or situation in the life of an object (or system) during which it satisfies some (invariant) condition, performs some activity or waits for some event. All objects of a given class that are in the same state must satisfy the same condition being implicitly presented by the state, react to the external stimulations in the same way and may undergo the same set of transitions.

A state vertex is an abstraction of a node in a statechart diagram. In general, it can be the source or destination of any number of transitions. In the abstract syntax for state machines, it is expressed as a subclass of ModelElement and includes several subclasses.

```
type Statevertex = \{ s : ModelElement • isStatevertex \}
value
  isStatevertex : ModelElement → Bool
```

Statevertex has four subclasses: State, Pseudostate, Synchstate and Stubstate.

```
type State = \{ s : Statevertex • isState(s) \},
  Pseudostate = \{ s : Statevertex • isPseudostate(s) \},
  Synchstate = \{ s : Statevertex • isSynchstate(s) \},
  Stubstate = \{ s : Statevertex • isStubstate(s) \}
value
  isState : Statevertex → Bool,
  isPseudostate : Statevertex → Bool,
  isSynchstate : Statevertex → Bool,
  isStubstate : Statevertex → Bool
```

A state has an optional entry action that is executed whenever the state is entered, an optional exit action that is executed whenever the state is exited, an optional doActivity activity that is executed while being in the state, an optional set of internal transitions that occur without exiting or entering this state if the corresponding events are triggered, and a deferrable event which is a list of events whose occurrence in the state is postponed until a state in which the listed events are not deferred becomes active.

These functions are defined as follows, where the type Event and Transition are defined as subtypes of ModelElement.

```
object
  A : ACTION

value
  isEvent : ModelElement → Bool,
  isTransition : ModelElement → Bool,
  entry : State → A.Action-set,
  exit : State → A.Action-set,
  doActivity : State → A.Action-set,
  internalTransition : State → Transition-set,
  deferrableEvent : State → Event-set
```

A pseudostate is an abstraction that encompasses different types of transient vertices in the statechart diagram and is used to connect multiple transitions into more complex transition paths.

```
type Pseudokind = initial | deepHistory | shallowHistory |
                 | join | fork | junction | choice
```

For every pseudostate, the function kind returns its kind.

```
value
  kind : Pseudostate → Pseudokind
```

A synchronous state is used for synchronizing concurrent regions in a state machine. It is used in conjunction with forks and joins to insure that one region leaves a particular state or a set of states before another region can enter a particular state or a set of states. Every synchronous state has a bound attribute which is a positive integer or has the value "unlimited" specifying the maximal count of the synchronous state. The count is the difference between the number of times the incoming and outgoing transitions of the synchronous state are fired.

```
value
  bound : Synchstate → Nat
```

A stubstate can appear within a submachine state to refer to an actual subvertex contained within the referenced state machine. The function referenceState designates the referenced state as a name formed by the concatenation of the name of a state and the successive names of states that contain it up to the top state.
type
value
name : State → Name,
referenceState : Substate → Name

The type State can be further specialized into three subtypes: Simplestate, Compositestate and Finalstate.

type
value
Simplestate = \{s : State | isSimple(s)\},
Finalstate = \{s : State | isFinal(s)\},
Compositestate = \{s : State | isComposite(s)\}
isSimple : State → Bool,
isFinal : State → Bool,
isComposite : State → Bool

A state vertex may be contained in a composite state. The function container returns the smallest composite state (if exists) that contains the state vertex:

value
container : Statevertex ⊇ Compositestate

Conversely, the function subvertices returns the set of direct substates being owned by a composite state.

value
subvertices : Compositestate → Statevertex-set

There are two Bool-valued attributes for a composite state: isConcurrent and isRegion. isConcurrent specifies whether the composite state is decomposed directly into two or more orthogonal conjunctive components called regions. isRegion indicates whether a composite state is a substate of a concurrent state.

value
isConcurrent : Compositestate → Bool,
isRegion : Compositestate → Bool

3.2 Transitions

A transition is a directed relationship between a source state vertex and a target state vertex. It may be part of a compound transition, which takes the state machine from one state configuration to another one reacting to a particular trigger event and when specified guard conditions are satisfied. Every transition has an optional trigger event which fires the transition; an optional guard which is a Bool-valued predicate that is evaluated at the time the event is dispatched and must be true to make the transition eligible to fire; an optional action to be performed when the transition fires which may act on the object that owns the state machine or other objects that are visible to the object; a source and a target state which designate the state being

the origination of the transition and the state that is active after the completion of the transition ¹.

type
value
Guard : Event
expression : Bool
source : Transition → Statevertex,
target : Transition → Statevertex,
trigger : Transition → Event-set,
guard : Transition → Guard-set,
effect : Transition → Action-set

axiom
∀ t : Transition
\card(trigger(t)) ≥ 0 \land \card(guard(t)) ≤ 1,  
∀ t : Transition
\card(effect(t)) ≥ 0 \land \card(guard(t)) ≤ 1,

∀ t : Transition
\card(trigger(t)) ≥ 0 \land \card(guard(t)) ≤ 1

Two functions outgoing and incoming are defined on Statevertex and returns the set of transitions with the state vertex as their source or target.

value
outgoing : Statevertex → Transition-set,
incoming : Statevertex → Transition-set

3.3 State Machines

A state machine is a specification which describes all possible behaviors of some dynamic model element. Behavior is modeled as a traversal of a graph of state nodes interconnected by transition arcs that are triggered by the dispatching of series of event instances. It specifies the sequences of states an object goes through during its lifetime together with its responses to events.

Every state machine has exactly one top state which designates the top level state that is a composite state at the root of the state containment hierarchy, and a set of transitions in the top state.

We define a function allsubvertices which returns the set of sub vertices contained in a composite state.

value
allsubvertices : Compositestate → write vs Unit

axiom
∀ cs : Compositestate
allsubvertices(cs) ≡
if isSimple(cs) then
vs := \{\}
else
vs := subvertices(cs);
finish := \{\};
while (v ∈ vs \land isComposite(v) \land v \notin finish) do
vs := vs \cup subvertices(v);
finish := finish \cup \{v\}
end
end

¹A transition may also have a set of sources or targets for the fork and join vertices. Here we restrict to the situation that one transition have only one source and one target state, which are obtained by the application of the function source and target respectively to the transition.
Another function \texttt{allsubvertexsInc} is defined to get all the subvertices of a composite state including the composite state itself.

\begin{verbatim}
value
  allsubvertexsInc : Compositestate \rightarrow write vs Unit
axiom
  \forall cs : Compositestate
  \cdot allsubvertexsInc(cs) \equiv allsubvertexs(cs); vs := vs \cup \{cs\}
\end{verbatim}

The type \texttt{Statechart} is defined as follows:

\begin{verbatim}
value
  top : StateChart \rightarrow State

Applying the function \texttt{allsubvertexsInc} to the top state of a statechart, we can get all the states of the statechart diagram.

For every state machine, the function \texttt{transitions} is defined to get the set of transitions owned by the state machine. The internal transitions are owned by their containing states and not by the state machine.

\begin{verbatim}
value
  transitions : StateChart \rightarrow Transition-set,
  internalTransition : State \rightarrow Transition-set
\end{verbatim}

Every state machine specifies the behavior of some model element. A model element may have several state machines although one is sufficient for most purposes, but every state machine specifies at most one model element. The function \texttt{context} returns the model element whose behavior is described by the state machine.

\begin{verbatim}
value
  context : StateChart \rightarrow Context-set
\end{verbatim}

The type \texttt{Submachinestate} is defined as a subtype of \texttt{Compositestate} being used as a syntactical convenience which facilitates reuse and modularity. It is semantically equivalent to a composite state.

\begin{verbatim}
value
  isSubmachine : Compositestate \rightarrow Bool
\end{verbatim}

For a submachine state, the function \texttt{submachine} gives a state machine that is to be substituted in place of the submachine state.

\begin{verbatim}
value
  submachine : Submachinestate \rightarrow StateChart
\end{verbatim}

Finally, the overloaded function \texttt{statemachine} is defined to return the state machine to which a state or transition belongs.

\begin{verbatim}
value
  statemachine : Transition \rightarrow StateChart,
  statemachine : State \rightarrow StateChart
\end{verbatim}

### 3.4 Well-Formedness Rules

The well-formedness rules (WFRs) for the model elements described above can be formalized in RSL.

Two auxiliary functions are defined before we touch the WFRs. The function \texttt{ancestor} checks whether one state is the ancestor of another in the hierarchy of containing relation. The function \texttt{LCA} returns the least common ancestor of two states.

\begin{verbatim}
value
  ancestor : State \times State \rightarrow Bool,
  LCA : State \times State \rightarrow State

axiom
  \forall s1, s2: State
  \cdot ancestor(s1, s2) \equiv
  \begin{cases}
    \text{true} & \text{if } s2 = s1 \\
    \text{true} & \text{if } \text{isEmpty}(container(s1)) \\
    \text{false} & \text{else if } \text{isEmpty}(container(s2)) \\
    \text{else ancestor(s1, container(s2))}
  \end{cases}

  \forall s1, s2: State
  \cdot LCA(s1, s2) \equiv
  \begin{cases}
    s1 & \text{if } \text{ancestor(s1, s2)} \\
    s2 & \text{else if } \text{ancestor(s2, s1)} \\
    \text{else LCA(container(s1), container(s2))}
  \end{cases}
\end{verbatim}

All the WFRs are represented as RSL axioms. A complete set of the WFRs can be found in appendix.

### 4 Formal Semantics of Statecharts

The semantics of UML statecharts are described in [11] by exploiting the UML meta-models. However, this approach fails to give a precise interpretation of nontrival statecharts structures. This drawback has motivated a challenging and active area for the precise definition of the formal semantics for UML statecharts. A variety of related work can be found in the literature, see e.g. [8, 6].

Once the abstract syntax of elements of UML statechart diagrams and well-formedness rules are precisely specified
via the RSL specification language, a precise semantics can also be given. As we have known, RSL is a specification language integrating both algebraic and coalgebraic techniques, which makes it a proper language for specifying the formal semantics of both class diagrams and statechart diagrams. A formal representation of the semantics for given elements in RSL paves a way to specify properties of systems being developed and to reason about their correctness.

The complete definition of coalgebraic semantics for statecharts is still an ongoing work. Therefore, here we will not give the detailed semantic definition for statechart diagrams, but only the general representation of statechart in RSL.

In [5], the semantic foundations of UML class diagrams has been given as a set of RSL templates, which forms the basis for the automatic translation mechanism from UML notations into RSL specification.

In general, a UML statechart diagram represents the evolution of objects of a given class, which can be interpreted as a coalgebra and specified via RSL specification. Therefore, the semantics of statecharts can be defined as RSL specifications. The abstract syntax is encoded according to the previous section and the wellformedness rules are given as axioms in the corresponding specification.

5 Case Study: Dynamic Host Configuration Protocol

In this section we present a case study on the Dynamic Host Configuration Protocol (DHCP) and show how to apply the formalization framework for development of systems. DHCP is now widely used in networks to provide configuration parameters to Internet hosts. Here we reduce the complex protocol to a simpler but expressive enough one which can be represented in a few UML diagrams.

DHCP is built on a client-server model where designated DHCP server hosts allocate network addresses and deliver configuration parameters to the dynamically configured hosts. For a host in a network communicating with other hosts in the network, a family of configuration parameters (for example, its unique network address, addresses of necessary servers and services) have to be set by its user or by a system administrator. The most important parameter is the IP address. DHCP supports three mechanisms for IP address allocation: automatic, dynamic and manual allocation. In automatic allocation, DHCP assigns a permanent IP address to a client. However, since the number of IP addresses is limited in a subnet (at most 254 network addresses per subnet), it might be impossible to allocate a static address to every client. Therefore, the dynamic allocation mechanism is also provided. In a dynamic allocation, DHCP assigns an IP address to a client for a limited period of time (or until the client explicitly relinquishes the address). Such a mechanism allows to automatically reuse an IP address when it is no longer needed by the client to which it was assigned. DHCP can also be simply used to convey the address assigned by the network administrator to the client in the manual allocation mechanism. In a particular network, one or more of these mechanisms can be used, depending on the policies of the network administrator.

In Figure 1 we show an example of such a network which has two servers and three clients. Each server can allocate one IP address to a client, that means, their pool of IP addresses is a singleton set.

Figure 1. A Network with two DHCP servers and three clients

5.1 UML Models for DHCP

In Figure 2 we show the functionalities of the network with respect to the DHCP protocol and the actors activating these functionalities. Two actors SysAdmin and User are shown in the diagram. For SysAdmin, there are two use cases: Initiate new DHCP server and Initiate new DHCP client. For User, there are
two use cases: Connect to network and Disconnect from network. Here we just briefly describe them as follows:

- **Initiate new DHCP server**: A new DHCP server can be created, with the information about its corresponding subnet of the network addresses, the lower range and the upper range of the block of IP addresses being provided.

- **Initiate new DHCP client**: A new DHCP client can be created by the system administrator.

- **Connect to network**: A user can initiate a connection of its host (client) to the network, which may be successful or unsuccessful.

- **Disconnect from network**: A client can be disconnected from the network by its user.

Generally, every use case can be realized as a composition of several atomic use cases which present the one-step system actions. Such compositions can be represented by sequence diagrams or collaboration diagrams. For example, for one scenario of use case **Connect to Network**, the sequence diagram is given in Figure 3 which shows the successful connection of a client to network. In fact, the use case **Connect to Network** can be refined by several use cases, each of which corresponds to a scenario and can be described by a sequence diagram, as shown in [18].

![Sequence Diagram](image)

**Figure 3. Sequence Diagram for one scenario of use case Connect to Network**

The class diagram given in Figure 4 shows the necessary classes and their relations. In this diagram, class **DHCP_Server** represents the set of DHCP servers that are part of a network. The private attributes **subnet**, **isBlock_lower** and **isBlock_upper** are used for the necessary parameters of a DHCP server: its corresponding subnet of the network addresses, the lower range and the upper range of the block of IP addresses.

**Class DHCP_Client** represents the set of DHCP clients that can connect and disconnect from the network via its public methods **connect** and **disconnect**. The information of the allocated IP address and the corresponding server are given in the two private attributes **myIPAddress** and **myServer**. A DHCP client can be activated by the operation **initiate**. The operations **offer**, **ack** and **nak** present the messages **DHCP_OFFER**, **DHCP_ACK** and **DHCP_NAK** that can be received.

**Class IPAddress** represents the IP addresses that are elements of the servers’ block of IP addresses. The relationship between servers and IP addresses is modeled by the composition relationship between the two classes.

**OCL** constraints are attached to the classes, attributes and methods in the class diagram to give the invariants for the classes and attributes, and the pre- and post-conditions for methods that must be satisfied. For example, in the class **IPAddress**, we have the pre- and post-conditions for the operations **create**, **bindIPAddress**, **unbindIPAddress** and **getBinding** as follows:

```
context IPAddress::create(ip: String): IPAddress
  post: result.bound = false

context IPAddress::bindIPAddress()
  pre: self.bound = false
  post: self.bound = true

context IPAddress::unbindIPAddress()
  pre: self.bound = true
  post: self.bound = false
```

![Class Diagram](image)

**Figure 4. Class Diagram for DHCP Model**
and the invariants on class DHCP_Client are given as:

context DHCP_Client
inv: self.aServer -> size <= 1
inv: self.network -> size = 1

Such constraints can be easily translated into RSL presentation by the pre- and post-conditions for the corresponding operations and axioms in RSL specifications.

The statechart depicted in Figure 5 specifies the behavior of IPAddress objects via a set of states that the objects can reside in and the transitions between them.

Figure 5. Statechart Diagram for IPAddress in DHCP Model

One interesting thing of the system is the concurrent connection to network conflicts. The system certainly allows many clients to connect to network and apply for IP addresses at the same time. However, one IP address can only be offered to at most one client at a given time. Broadcasting competition is the problem that can not be avoided. For example, when two clients send DHCP_DISCOVER messages to the same servers at a same time, it is possible that both of them receive the DHCP_OFFER message from the same server for the same IP address. Obviously, the IP address can only be given to the client whose request message is first received by the server. When the second client send its request message to the server, the IP address may not be available any more, and a DHCP_NAK message may be sent by the server. That means, the application of the IP address by the second client is not successful. If it does not reserve the IP address information provided by another DHCP_OFFER message, it perhaps could not be allocated successfully at next time because other clients may take the IP address. Therefore, a client may not be allocated an IP address for several times. To avoid such an unfairness and make the request for IP addresses fair to clients, we should introduce a reserved state for IPAddress class in the system design, as shown in the statechart diagram in Figure 5. This is one constraint on the Connect to Network use case, i.e., the system should guarantee that the client could definitely get the IP address if it is available when it delivers its request. Therefore, when an IP address is reserved, it is not free and can not be got by other clients.

On the other hand, we also need to avoid the bad case that IP addresses are always reserved but not allocated. Hence, the system should have a nonfunctional requirement: no client is allowed to reserve an IP address infinitely without using it. Otherwise, lots of IP addresses could not be available because they are reserved by a few clients but not available to other clients. Therefore, we make a temporal constraint that the server will collect those reserved IP addresses and make them free if they are reserved more than 5 minutes without being used.

5.2 RSL Specification of DHCP

Here we only give part of the translation from the statechart in Figure 5 to the corresponding RSL specification. Such a translation is based on the framework proposed in Section 3.

The statechart consists of a top state ip_top, three states Available, Reserved and Allocated and one initial state. All of them are specified as elements of type StateVertex, and the functions being used are defined by axioms.

<table>
<thead>
<tr>
<th>type</th>
<th>StateVertex == ip_top</th>
<th>Available</th>
<th>Reserved</th>
<th>Allocated</th>
<th>Init</th>
</tr>
</thead>
<tbody>
<tr>
<td>axiom</td>
<td>∀ s: StateVertex •</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>isPseudostate(s) ≡</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>if s = Init then true</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>else false</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>end</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>∀ s: StateVertex •</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>isState(s) ≡</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>if s = Init then false</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>else true</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>end</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>∀ s: StateVertex •</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>isSynchstate(s) ≡</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>false</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>∀ s: StateVertex •</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>isStubstate(s) ≡</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>false</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The transitions in the statechart diagram are specified as instances of type Transition.

<table>
<thead>
<tr>
<th>type</th>
<th>Transition == ip_getFreeIP</th>
<th>ip_offer</th>
<th>ip_release</th>
<th>ip_timeout</th>
<th>ip_request</th>
<th>ip_disconnect</th>
</tr>
</thead>
</table>

The relationship between the transitions and states can be stated by the functions source, target, outgoing and incoming. For example, for the transition ip_request and the statevertex Reserved, we have the following axioms.

| axiom    | source(ip_request) ≡ Reserved, |          | outgoing(Reserved) ≡ {ip_request}, |
|          | target(ip_request) ≡ Allocated, |          | incoming(Reserved) ≡ {ip_offer, ip_release, ip_timeout} |
6 Conclusion and Future Work

In this paper we presented a framework for the formaliza-
tion of UML statechart diagrams using the RAISE spec-
ification language RSL as the underlying formal founda-
tion. The main motivation of this work is to give a precise de-
scription of the UML statechart diagrams in RSL as the con-
tinuation of the previous work on formalizing UML class di-
agrams in RSL [5], and to provide a basis for the defini-
tion of the formal semantics for statecharts. Our approach 
integrates the formal specification language RSL and a non-
trivial subset of the semi-formal graphical language UML.
By translating the UML diagrams to RSL specifications, we 
can use RSL tools to analyze and refine the specifications 
precisely, and verify properties of the specifications.

In our approach, the RAISE specification language RSL 
is employed to formalize statecharts. Comparing with oth-
ers’ work, the method proposed in this paper is applicable 
for the formalization of not only UML statechart diagrams 
but also other view models (the formalization of class dia-
grams in RSL has been given in [5]). Therefore, our work 
makes it possible to integrate different view models in a uni-
fying formal framework.

Appendix: Well-Formedness Rules for State-
charts

∀ s : Compositestate • 
∀ b : Statevertex-set • 
( b ⊆ subvertexs(s) ∧ (∀ cs : Statevertex • cs ∈ b ⇒ 
isPseudostate(cs) ∧ kind(cs) = initial)) ⇒ card b ≤ 1, 
∀ s : Compositestate • 
∀ b : Statevertex-set • b ⊆ subvertexs(s) ∧ (∀ cs : Statevertex • cs ∈ b ⇒ 
isPseudostate(cs) ∧ kind(cs) = deepHistory) ⇒ card b ≤ 1, 
∀ s : Compositestate • 
∀ b : Statevertex-set • b ⊆ subvertexs(s) ∧ (∀ cs : Statevertex • cs ∈ b ⇒ 
isPseudostate(cs) ∧ kind(cs) = shallowHistory) ⇒ card b ≤ 1, 
∀ s : Compositestate • 
isConcurrent(s) ⇒ card(subvertexs(s)) ≥ 2, 
∀ s : Compositestate • 
isConcurrent(s) ⇒ (∀ cs : Statevertex • cs ∈ subvertexs(s) ⇒ isComposite(cs)), 
∀ s : Compositestate • 
∀ cs : Statevertex • cs ∈ subvertexs(s) ⇒ container(cs) = s), 
∀ s : Submachinestate • 
isConcurrent(s) = false, 
∀ s : Submachinestate • 
∀ ss : Statevertex • ss ∈ subvertexs(s) ⇒ isSubstate(ss), 
∀ t : Finalstate • 
\( \text{card(outgoing(t))=0} \), 
∀ p : Pseudostate • 
kind(p) = initial ⇒ 
\( \text{card(outgoing(p))≤1 ∧ incoming(p) = } \{ \}, \) 
∀ p : Pseudostate • 
kind(p) = deepHistory ∨ kind(p) = shallowHistory ⇒ 
\( \text{card(outgoing(p))≤1} \), 
∀ p : Pseudostate • 
kind(p) = join ⇒ 
\( \text{card(outgoing(p)) = 1 ∧ card(incoming(p)) ≥ 2} \), 
∀ p : Pseudostate • 
kind(p) = fork ⇒ 
\( \text{card(incoming(p)) = 1 ∧ card(outgoing(p)) ≥ 2} \), 
∀ p : Pseudostate • 
kind(p) = junction ⇒ 
\( \text{card(outgoing(p)) ≥ 1 ∧ card(incoming(p)) ≥ 1} \), 
∀ t : Transition • 
\( t ∈ \text{transition(p) ⇒ } \) 
\( \text{ isEmpty(top(statemachine(t)))) ⇒ } \) 
context(statemachine(target(t))) = context(statemachine(source(t))),
∀ p : State • 
isTop(p) ⇒ isComposite(p) ∧ isEmpty(container(p)) ∧ outgoing(p) = {}.
∀ s : Synchstate • 
bound(s) > 0,
∀ s : Synchstate • 
∀ t1, t2 : Transition • 
(t1 ∈ outgoing(s) ∧ t2 ∈ outgoing(s) ⇒ 
context(statemachine(target(t1))) = context(statemachine(target(t2))))
∀ p : Transition • 
kind(p) = join ⇒ (∀ t1, t2 : Transition • 
t1 ∈ outgoing(p) ∧ t2 ∈ outgoing(p) ∧ t1 ≠ t2 ⇒ 
isConcurrent(container(LCA(source(t1), source(t2))))),
∀ p : Transition • 
kind(p) = fork ⇒ (∀ t1, t2 : Transition • 
t1 ∈ outgoing(p) ∧ t2 ∈ outgoing(p) ∧ t1 ≠ t2 ⇒ 
isConcurrent(container(LCA(source(t1), source(t2))))),
∀ s : Pseudostate • 
kind(s) = fork ⇒ 
(∀ t : Transition • t ∈ incoming(s) ∧ t ∈ outgoing(s) ⇒ 
trigger(t) = {} ∧ guard(t) = {}),
∀ s : Pseudostate • 
kind(s) = join ⇒ 
(∀ t : Transition • t ∈ incoming(s) ∧ t ∈ outgoing(s) ⇒ 
trigger(t) = {} ∧ guard(t) = {}),
∀ s : Pseudostate • 
kind(s) = fork ⇒ 
(∀ t : Transition • t ∈ outgoing(s) ⇒ isState(target(t))),
∀ s : Pseudostate • 
kind(s) = join ⇒ 
(∀ t : Transition • t ∈ incoming(s) ⇒ isState(source(t))),
∀ s : Pseudostate • 
kind(s) = initial ⇒ 
(∀ t : Transition • 
\( t ) \text{ source(t) = s ⇒ } \) 
context(t) = context(t) = initial) ∧ trigger(t) = create 
\),
∀ s : StateChart • 
\( \text{card(context(s)) = 0 ∨ card(context(s)) = 1} \),
∀ s : StateChart • 
∀ c : Context • 
(c ∈ context(s) ⇒ isBehaviorFeature(c) ⇒ 
(∀ t : Transition • 
\( t ∈ \text{transitions(s)} ⇒ ) \) 
\( (isPseudostate(source(t)) ∧ kind(source(t)) = initial) ⇒ trigger(t) = {} \) )

References

ified Modeling Language User Guide. Addison Wes-
ley, 1999.

Models to Formal Specifications. In J. Bézivin and P.-
A. Muller, editors, The Unified Modeling Language,


