The SMUML UML subset

Tommi Junttila
Helsinki University of Technology TKK
Laboratory for Theoretical Computer Science
http://www.tcs.hut.fi/~tjunttil

Jori Dubrovin
Helsinki University of Technology TKK
Laboratory for Theoretical Computer Science
email: Jori.Dubrovin@tkk.fi

December 20, 2007

Abstract

This document describes the UML subset developed and used in the project "Symbolic Methods for UML Behavioural Diagrams" (SMUML). It also gives a formal semantics for the subset. The semantics includes a formal definition for the structure and behavior of a class of hierarchical UML state machines.

1 Introduction

This document describes the UML subset developed and used in the project "Symbolic Methods for UML Behavioural Diagrams" (SMUML). The SMUML project, carried out at the Laboratory for Theoretical Computer Science of Helsinki University of Technology during the years 2005–2007, has been funded by the Finnish Funding Agency for Technology and Innovation (TEKES), Nokia Research Center, Conformiq Oy, and Mipro Oy. The goal of the project has been to develop new symbolic methods for analyzing the dynamic behavior of UML models. As UML is a very large modelling language, a subset of UML that is supported by the tools developed in the project has been defined. The main design goals for the subset are that it should be (i) suitable for modelling systems composed of asynchronously executing objects communicating with each other though attribute access and message passing, and (ii) amenable for automatic analysis by means of model checking [2] tools. As a result, a UML system in the proposed UML subset is composed of a finite set of asynchronously executing objects that are instances of classes in the UML model. The objects can communicate with each other via message passing and attribute access. The behavior of each active object is described by the hierarchical state machine associated with the class of the object. The guards and effects appearing in the transitions of the state machines are written in a Java-like action language Jumbala [4]. In addition to specifying a UML subset, this document also gives a formal semantics for the subset. In particular, the semantics includes a formal

definition of the structure and the behavior of a class of hierarchical UML state machines.

To specify the subset of UML, we use the UML version 1.4 [6] as the starting point. The reason for using a rather old version of UML is a historical and practical one: when the SMUML project was started, the easiest way to access and manipulate UML models was to use the Python programming language interface of the Coral meta-modeling toolkit [1, 3] developed at the Åbo Akademi. At that time, as well as today, Coral only had graphical editor for UML 1.4, not for later versions. On the positive side, as UML 1.4 is basically a subset of later versions of UML, the techniques developed in the project either directly apply or can potentially be extended to more recent versions of UML. Of course, constructs like composite structures and ports may pose some research challenges but, for instance, the semantics for state machines described in this document should be usable as a basis for semantics of UML 2.0 state machines.

The rest of this document is divided in two parts:

- First, Section 2 gives the structural characterization of the models in the proposed UML subset.
- Second, Section 3 gives a formal semantics for the models in the subset. In particular, the semantics of the state machines are described in detail.

1.1 Model Input Format and a Model Validation Tool

The tools developed in the project use the Coral meta-modeling toolkit [1, 3] developed at the Åbo Akademi to access the UML models. They expect the models to also conform to the UML 1.4 meta-model as implemented in Coral and to be given in the XMI file format as supported by Coral.

The SMUML project software tool set contains a model validator tool that checks whether the given Coral UML 1.4 model conforms to the SMUML UML subset. It can be invoked with

```
python validate_model.py umlmodel.xmi
```

where `umlmodel.xmi` is the model to be validated.

1.2 Acknowledgements

The financial support of the Finnish Funding Agency for Technology and Innovation (TEKES), Nokia, Conformiq, and Mipro is gratefully acknowledged. The authors also wish to thank Antti Huima, Toni Jussila, Timo Latvala, and Heikki Tauriainen for their ideas, feedback, and comments during the development of the UML subset.

2 Structure

This section describes the structure of the UML (1.4) models conforming to the SMUML UML subset. It is required that the reader is familiar with the UML 1.4 meta-model specification [6]. In order to improve readability, the UML concepts such as classes, attributes, and associations in the UML meta-model are written in a different font in this Section; for instance, SimpleState for the

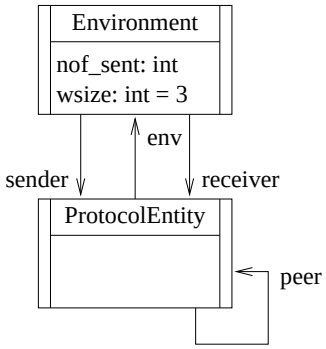

Figure 1: Classes

UML 1.4 meta-class "SimpleState" or s.outgoing for the association "outgoing" of an element s (that is an instance of StateVertex).

2.1 Models

A model in the SMUML UML subset can contain Packages, Classes, DataTypes, and Signals. If the model is to be simulated or analyzed with model checking tools, it must also contain Objects and Links describing the initial configuration of the modeled system (see Section 2.9).

2.2 Packages

Packages can be used to decompose models in smaller parts. They can contain other Packages, Classes, DataTypes, and Signals. Currently they carry no semantic or namespace information.

2.3 Data Types

The primitive, non-reference data types in a model are represented by DataTypes. Currently the only supported sub-type of DataType is Primitive. Each Primitive d must be owned by the model or by a Package. The following Primitives are allowed:

- If d.name is "boolean", then d represents the standard Boolean data type with the domain $\{false, true\}$. Other attributes of d are ignored.
- If d.name is "int", then d represents the standard 32-bit signed integer data type with the domain $\{-2^{31}, \ldots, -1, 0, 1, \ldots, 2^{31} - 1\}$. Other attributes of d are ignored.

2.4 Classes

The name of a Class c, c.name, must be a string matching the regular expression "[a-zA-Z_][a-zA-Z_0-9]*". In addition, as the Packages provide no namespace information at the moment, the names of the Classes in a model must be disjoint. Furthermore, inheritance and aggregation are not supported at the moment.

Example 2.1 Figure 1 shows two Classes, "Environment" and "ProtocolEntity". It is common, although not required, to capitalize Class names. ♣

2.4.1 Non-reference Attributes

The primitive, non-reference attributes of a Class c are declared by the Attribute elements in c.feature. For each Attribute a in c.feature,

- a.changeability must be changeable, implying that constant attributes cannot be defined currently.
- a.initialValue is an Expression whose body is either empty or a constant Jumbala expession of type matching a.type, refer to Section 2.11 for Jumbala expressions.
- a.name must be a string matching the regular expression "[a-zA-Z_][a-zA-Z_0-9]*".
- a.multiplicity must be 1..1.
- a.ordering must be unordered.
- a.ownerScope must be instance, implying that a is a so-called instance attribute (each instance of the Class owns its own copy of the Attribute); class (i.e. static in C++ terms) attributes cannot be defined currently.
- a.targetScope must be instance.
- a.type must be a DataType (see Section 2.3).
- a.visibility must be public.

When an instance (object) of a Class is created when executing the model, the value of each Attribute a of the object is initialized by the following rules.

1. If the object is created in the beginning of the execution because the model contains it as an Object o (see Section 2.8) and o contains an AttributeLink l for the Attribute in o.slot, then a is initialized to have the value l.value.
2. If the previous case does not apply but the body of a.initialValue is not empty, then a is initialized to have that value.
3. Otherwise a is initialized to have the default value of a.type, i.e. 0 for the Primitive "int", and false for the Primitive "boolean".

Example 2.2 The Class "Environment" in Figure 1 has an Attribute "nof_sent" that is of the primitive type "int" and has no initial value definition. The initial value of the Attribute "wsize" of the class is defined to be "3". ♣

2.4.2 Reference Attributes: Associations

Object reference attributes are modeled by using Associations. Only unidirectional associations are supported. Each Association a must have two AssociationEnds in a.connection. The so-called source AssociationEnd e_1 must have

- e_1.multiplicity of 0..*,
- no name,
- e_1.isNavigable set to false, and
- e_1.participant referencing to a Class.

The so-called target AssociationEnd e_2 must have

- e_2.multiplicity of 0..1,
- e_2.name matching the regular expression "[a-zA-Z_][a-zA-Z_0-9]*" (this is the name of the reference attribute in question, the name of the Association is not relevant or even required),
- e_2.isNavigable set to true, and
- e_2.participant referencing to a Class.

The Association a thus defines that the Class e_1.participant has a reference attribute called e_2.name referencing to an object of type e_2.participant. In addition, for both AssociationEnds, aggregation must be "none" and visibility "public". Like non-reference attributes, all the reference attributes are instance attributes, not class (i.e. static in C++ terms) attributes.

The names of the attributes (including both non-reference and reference attributes) of a class must be disjoint.

Example 2.3 The Class "ProtocolEntity" in Figure 1 has a reference attribute called "env". In an instance (object) of the Class, the value of the attribute is either null or a reference to an instance of the Class "Environment". ♣

2.4.3 Operations

Operations or methods are not supported at the moment.

2.4.4 Active Classes and State Machines

A Class c is active if the attribute c.isActive is true. If c is active, then (and only then) c.behavior must contain exactly one StateMachine sm that describes the behavior of an instance of the Class. The StateMachine sm must be owned by the Class c, i.e. it must also belong to c.ownedElement. Requirements for StateMachines are described in Section 2.7.

2.5 Signals

Signals define the signatures of the messages sent between the instances of the active Classes.

Each Signal *sig* must have a name in *sig*.name that is a string matching the regular expression "$?[a-zA-Z_][a-zA-Z_0-9]*"; e.g. "tick" and "$tack" are valid signal names. The names of the Signals in a model must be disjoint. If the Signal name starts with the dollar sign, then the signal is interpreted as a so-called *external signal* that are received non-deterministically from the environment.

The parameters of a Signal *sig* are described in *sig*.feature that is an ordered set of Attributes. Each Attribute a in *sig*.feature must have a.type that is either a DataType (see Section 2.3) or a Class (see Section 2.4). The parameters of DataType type are passed by value while the ones of Class type are passed by reference. The multiplicity in a.multiplicity must be 1..1 if a.type is a DataType and 0..1 if a.type is a Class (null reference is allowed). External signals may not have parameters.

2.6 Signal Events

SignalEvents are used in the transition triggers of state machines to model signal (message) reception and in the states of state machines to model deferring of signals (messages).

For each SignalEvent e, the attribute e.signal must reference to a Signal (see Section 2.5). The attribute e.parameter must contain as many Parameters as the Signal e.signal has Attributes in e.signal.feature. Each Parameter p in e.parameter should have a p.name, and the Class c owning the SignalEvent e must have a (primitive or reference) attribute with the name p.name and of the same type as the type of the corresponding Parameter in the Signal e.signal. The names of the Parameters in e.parameter must be disjoint. When a signal (message) is consumed by a transition, the parameter values are assigned to the attributes of the executing object.

2.7 State Machines

Each active Class in the model is attached with a StateMachine describing its dynamic behavior. In the specified UML subset, the transitions in state machines are either triggered by SignalEvents or have no trigger (i.e. are completion transitions); method calls and remote procedure calls via CallEvents are thus not supported in the UML subset. Each active object has a first-in-first-out (FIFO) *input queue* for messages sent to the object as well as an internal FIFO *defer queue* for messages that are temporarily deferred.

A StateMachine *sm* can contain SimpleStates, FinalStates, concurrent and non-concurrent CompositeStates, initial PseudoStates, and choice PseudoStates. The top state *sm*.top of the state machine must be a non-concurrent CompositeState. Each non-concurrent composite state must contain exactly one initial Pseudostate. If a non-concurrent composite state has a concurrent composite state as its container, then it is called a region. Each concurrent composite state must have at least two non-concurrent composite states in its subvertex set (i.e. as its regions) and may not have any subvertices of other types. If two states are orthogonal, then they are not allowed to be interested in the same

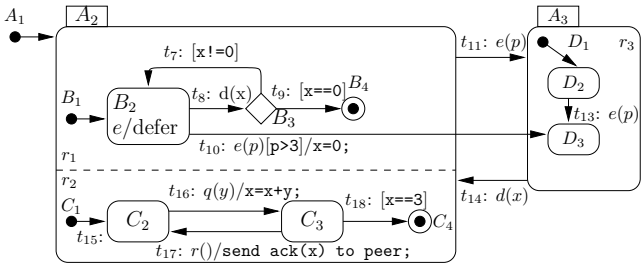

Figure 2: A UML state machine.

signals. That is, the signals that (i) are deferrable in the states or (ii) serve as triggers in transitions leaving the states must be disjoint.

Example 2.4 In the StateMachine in Figure 2, A_1 is an initial Pseudostate, A_2 is a concurrent CompositeState with two regions r_1 and r_2, B_2 is a SimpleState, B_3 is a choice Pseudostate, and B_4 is a FinalState. The SimpleStates B_2 and C_3 are orthogonal. ♣

2.7.1 Transitions

Each Transition t in a StateMachine sm is owned by sm and listed in sm.transitions.

First, the trigger t.trigger must either be absent or a SignalEvent (recall Section 2.6). If there is no t.trigger, then the transition is called a completion transition, otherwise a signal triggered transition. In UML, the completion transitions are though to be triggered by the implicit completion event. The transitions leaving a PseudoState must be completion transitions; and there must be at least one such transition; in addition, it is required that in each possible global configuration of the model, there is at least one outgoing transition whose guard evaluates to true.

Example 2.5 In Figure 2, the transition t_9 leaving the choise pseudostate B_3 is a completion transition. The transition t_{10} is triggered with the signal e; when t_{10} is fired, the first parameter of the received message (instance of the signal e) is assigned to the attribute p of the executing object. ♣

The guard t.guard of a Transition t must either be absent or a Guard with t.guard.expression.body including a side-effect free Jumbala expression. If there is no t.guard, then the guard is implicitly always true. The guards are evaluated in a global configuration where the possible message parameters (recall Section 2.6) have been assigned to the attributes of the owning class as explained in Section 2.6).

Example 2.6 In Figure 2, the guard of transition t_8 is implicitly true. The guard of transition t_{10} is a Jumbala expression "p>3" and therefore the transition t_{10} is enabled only when the state configuration is in a stable configuration where the source state B_2 is active, the first message in the input queue is an instance of the signal e, and the first parameter of the message if greater than 3. ♣

If a Transition t has an effect, it is given in t.effect. If t.effect exists, it should be an instance of either Action or UninterpretedAction such that (i) both t.effect.recurrence.body and t.effect.target.body are empty strings, and (ii) t.effect.script.body is the Jumbala code for the effect.

Example 2.7 In Figure 2, the effect of the transition t_{10} is the statement "`x=10`". ♣

2.7.2 Deferrable Events

A SimpleState or CompositeState s can have SignalEvents in s.deferrableEvent denoting that, when the state is active in a stable state configuration, the first message in the input queue, if it is of that Signal type, can be deferred if it is not explicitly consumed by a transition leaving from that state or any of its active substates. Deferring means that the message is removed from the input queue and put in the defer queue. When the state machine reaches the next stable state configuration after firing some transition(s), the defer queue is flushed to the front of the input queue. External signals (cf. Section 2.5) may not be deferrable.

Example 2.8 If the state machine in Figure 2 is in the stable state configuration $\{A_2, B_2, C_2\}$ and the first message in the input queue is $e(2)$, then the transition t_{10} is not enabled and the message is deferred. ♣

2.7.3 Execution Granularity and Steps

In order to apply model checking techniques for analyzing the dynamic behavior of UML systems, we have to fix the granularity level of executions. That is, we have to decide what consitutes to an atomic step. In the proposed UML subset we consider one step to consists of exactly one object executing one of the following actions:

- Firing one transition in its state machine. This includes evaluating the transition guard, copying the signal parameters to the corresponding attributes, executing the code in the transition effect, changing the state configuration of the state machine, and flushing the defer queue in front of the input queue if the new state configuration is stable.
- Implicitly consuming the implicit completion event.
- Deferring the first message in the input queue.
- Implicitly consuming the first message in the input queue.

2.7.4 Flat and Normalized State Machines

A state machine is *flat* if the only composite state it contains is the top state. Each state machine can be transformed into a corresponding flat one by basically enumerating its possible state configurations.

A flat state machine is *normalized* if for each SimpleState it holds that either

- all outgoing transitions are signal triggered,

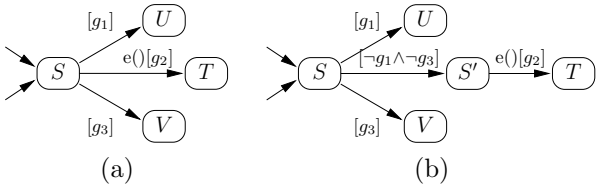

Figure 3: A part of a state machine (a) and its normalized version (b).

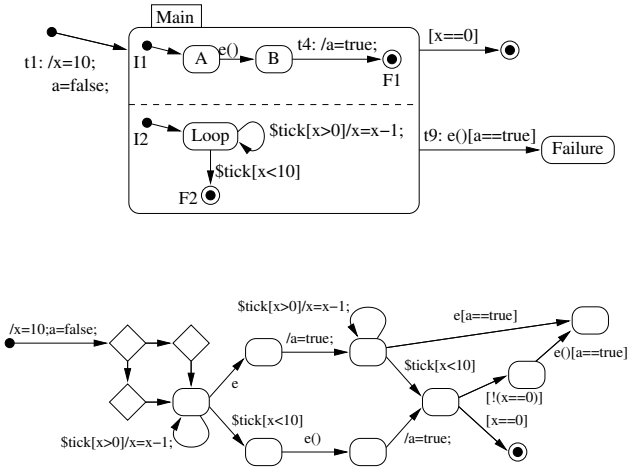

Figure 4: A hierarchical state machine and its flat and normalized version.

- all outgoing transitions are completion transitions and it is guaranteed that, in every possible global configuration of the system, at least one their guards will evaluate to true, or
- there are no outgoing transitions.

Due to the second condition, the concept of implicit consumption of a completion events (i.e. quiescing) is not needed for normalized flat state machines. It is possible to transform a non-normalized state machine to a corresponding normalized version while preserving (i) deadlocks, (ii) assertion violations, and (iii) stuttering invariant linear time temporal logic properties. This is done by making the quiescing steps explicit by making duplicate states for completion sensitive states and extra completion transitions for the quiescing case. This construction is illustrated in Figure 3.

The state machine hierarchy flattening tool in the SMUML project software tool set, `flatten_state_machine_hierarchy.py`, performs both hierarchy flattening and the normalization transform, thus producing flat and normalized state machines. For instance, Figure 4 shows a hierarchical state machine and its flat and normalized version.

2.8 Objects, Slots, and Links

The Objects in a model are used to describe the initial global configuration of the model (see Section 2.9). An Object o is an instance of a Class c in the model.

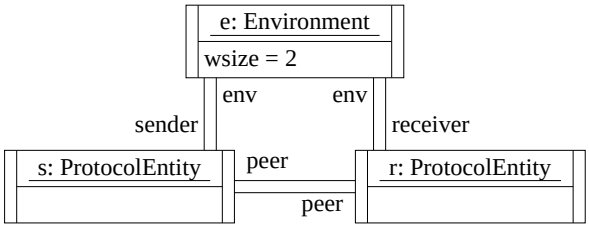

Figure 5: Objects and links.

Thus o.classifier must contain c (and nothing else).

Example 2.9 Figure 5 shows a global configuration composed of three Objects: e, s, and r. The Object e is an instance of the Class "Environment". ♣

Initializing Primitive Attributes. The initial values of non-reference attributes in an Object o can be specified by including AttributeLinks in o.slot (recall the attribute value initialization rules in Section 2.4). For each AttributeLink l in o.slot,

- l.attribute must be a non-reference Attribute of the Class o.classifier,
- l.instance must be o, and
- l.value must be a DataValue v such that
  - v.classifier is the DataType l.attribute.type,
  - v.name describes a member in the domain of v.classifier. That is, it is (i) the string representation of an integer in the range $[-2^{31}, \ldots, 2^{31}-1]$ when v.classifier is the Primitive "int", or (ii) the string "false" or "true" when v.classifier is the Primitive "boolean".

  A DataValue can be owned by an Object, a Class, a Package, or the model.

The Attributes of AttributeLinks in o.slot must be disjoint, i.e. it is not allowed to define the initial value of an Attribute multiple times.

Example 2.10 In the global configuration shown in Figure 5, the Attribute "wsize" of the object e is initialized to 2. Without the slot in the Object definition, it would have been initialized to 3 because of the initial value description in the definition of the Class "Environment" in Figure 1. The Attribute "nof_sent" of the Object e is initialized to the default value 0 of the "int" data type as it has no slot or initial value definition. ♣

Initializing Reference Attributes. The initial values of reference attributes (as described by Associations, recall Section 2.4.2) in Objects can be specified by Links. For each Link l the following must hold.

- l.association is the Association that the link is initializing.

- l.connection must contain two LinkEnds correspoding to the AssociationEnds in l.association.connection. For both LinkEnds e, e.associaionEnd is the corresponding AssociationEnd and e.instance is an Object matching the type of the AssociationEnd, i.e. e.instance.classifier should equal to e.associaionEnd.participant.

The initial value of a reference attribute must not be defined multiple times by Links.

Example 2.11 In Figure 5 the reference attribute "env" of the Object s is initialized by a Link to refer to the Object e. ♣

2.9 Initial Global Configuration

The initial global configuration of the model describes one instantiation of the model and serves as the starting point for the dynamic analysis methods such as simulation and model checking. It is composed of the Objects (refer to Section 2.8) included in the model, describing the objects existing in the beginning of the execution of the system. The non-reference attributes of the objects in the initial global configuration are initialized as described in Sections 2.4 and 2.8. The reference attributes of the objects are initialized by the Links in the model, see Section 2.8. The input and defer queues (recall Section 2.7) of active objects (instances of active Classes) are empty in the initial global configuration.

Example 2.12 Figure 5 shows an initial global configuration explained in more detail in Section 2.8. ♣

2.10 Comments and Tagged Values

Each elements e in a model can be associated with Comments in e.comment and TaggedValues in e.taggedValue. They have no general semantic meaning but can be used, for instance, by the modeller to annotate the model with comments or by model transformation tools to pass information to other tools.

2.11 Action Language

The guards and effects of the transitions in a model are described with the Java-like action language Jumbala [4]. From the perspective of the action language statements and expressions, each Class in a model corresponds to a Jumbala class of the same name and attributes. Non-reference attributes of classes are mapped in a one-to-one way from UML to Jumbala, e.g. if a UML Class has a primitive attribute x of type "int", then the corresponding Jumbala class also has an attribute x of Jumbala type "int". Reference attributes are mapped in a similar way: if a UML Class A has a reference attribute x to a Class B, then the corresponding Jumbala class A has an attribute x referencing to the Jumbala class B. For more details, see [4].

The following subset of Jumbala is supported in the models currently.

- Attribute reference.

  The attributes in the enclosing and other objects can be referenced in the standard way, e.g. the expression "`x`" refers to the attribute x of the

enclosing object and "`a.y`" refers to the attribute y of the object referenced by the reference attribute a of the enclosing object.

- Operations.

  Standard logical and arithmetic operations can be applied in expressions; for instance, "`x + 3 < 10`" is a legal Boolean expression provided that x is an "int" attribute. That is, usual typing constraints apply.

  An expression is side-effect free if evaluating it does not cause changes in the state of the system. For instance, creation of new objects with the `new` construction as well as the pre- and postfix increment and decrement operations (such in x`--`) are not allowed in side-effect free expressions.

- Assignments.

  An assignment statement is of form "r `=` e`;`", where r is a reference to an attribute and e is an expression. For instance, "`a.y = x+3;`" is an assignment statement. Both r and e must be side-effect free.

- Assertion statements.

  A statement of form "`assert(`e`);`", where e is a side-effect free Boolean expression, cause the execution to terminate in an error if the expression e evaluates to false.

- Send statements.

  For asynchronous message passing Jumbala includes send statements of form "`send` s`(`e_1`,...,`e_n`)` `to` t`;`", where (i) s is the name of a Signal in the model, (ii) $e_1, \ldots, e_n$ are expressions describing the values of the message parameters, and (iii) t is an expression referencing to the object to which the message is sent (also called the target of the send statement). External signals cannot be sent, i.e. s must not start with the dollar sign. The parameter expressions $e_1, \ldots, e_n$ as well as the target expression t must be side-effect free.

- Control flow constructs.

  In addition to sequences of statements, the following control flow constructs of Jumbala are allowed in the effects of transitions in the model: labels, `if`, `for`, `while`, `do`, `break`, and `continue`. For instance,

```
a:
while(true)
  for(i=0;i<3;i++) {
    if(i>j)
      break a;
    j = j - 1;
  }
```

  is allowed. The condition part of `if`, `for`, and `while` constructs must be a side-effect free Jumbala expression. In addition, the initialization and update parts of each `for` construct should consist of assignments of form "r `=` e" with r and e being side-effect free expressions. As an example, "`for(i = x++; i<3; i++) ...`" is not allowed.

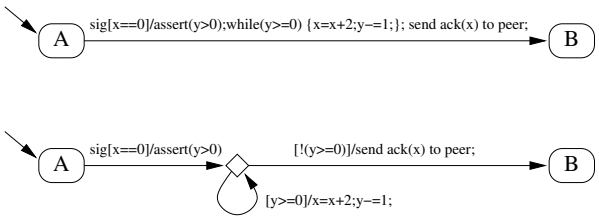

Figure 6: A transition with an action containing a "while" statement and the corresponding flattened version with an extra choice pseudo-state.

- Creation of new objects.

  New objects can be created with assignment statements of form

  ```
  r = new C();
  ```

  where r is a side-effect free reference expression and C is the name of a Class in the model. As method calls are not supported at the moment, neither are constructor methods.

Note that local variable declarations are not currently supported.

Note that not all tools developed in the SMUML project accept the full subset of action language described above. For instance, the symbolic model checking tools do not allow the control flow constructs to appear in the state machine transition effects. Fortunately, such constructs can be eliminated by inserting extra pseudo-states into the state machine, in a sense lifting the control flow structure from the action language level into the state machine level (see Figure 6 for an example). A tool, called `flatten_state_machine_action_language.py`, has been developed in the project for removing all "`if`", "`while`", "`do`", "`for`", "`break`", and "`continue`" statements from state machine transition effects. But please note that doing this transformation changes the behavior of the model as the transition is no longer atomic as it's code is now distributed over several transitions in the model, allowing other objects to execute while the transition is in one of its intermediate choice pseudostates. That is, the resulting model has more behaviors than the original one.

3 Formal Semantics

In this section we give a semantics for the proposed subset of UML models. The main focus is on the semantics of state machines; the semantics of the action language are essentially abstracted away as it should be intuitively clear.

3.1 Types, Signals, and Classes

As the main focus of the semantics definition is on UML state machines, other relevant parts of UML models, most notably data value manipulation by action language expressions and statements, are defined only in a very abstract level.

The set of all finite sequences over a set X is denoted by X^*. If $a = \langle a_1, \ldots, a_k \rangle \in X^*$, $b = \langle b_1, \ldots, b_l \rangle \in X^*$, and $x \in X$, then $concat(a, b) =$

$\langle a_1, \ldots, a_k, b_1, \ldots, b_l \rangle$, $dequeue(a) = \langle a_2, \ldots, a_k \rangle$ (undefined if $k = 0$, the empty sequence $\langle\rangle$ if $k = 1$), and $append(a, x) = \langle a_1, \ldots, a_k, x \rangle$.

Data Types and Action Language. To capture data types in UML models, a finite set $\mathcal{T}$ of *types* is assumed, each type $T \in \mathcal{T}$ being associated with a non-empty *domain* set $dom(T)$. In particular, the Boolean type $\mathbb{B}$ with $dom(\mathbb{B}) = \{false, true\}$ belongs to $\mathcal{T}$. A *typed variable* is a name x associated with a type $type(x) \in \mathcal{T}$. The guards and effects appearing in state machines are expressed with a strongly typed *action language* $\mathcal{L}$ over the types; $\mathcal{L}_{\mathbb{B}} \subset \mathcal{L}$ denotes the set of side-effect free Boolean valued expressions and $\mathcal{L}_{\mathrm{Stmt}} \subset \mathcal{L}$ the set of (possibly compound) statements in the action language.

Signals and Messages. As state machines can communicate with each other by sending messages, we assume a finite set *Sigs* of *signals*. Each signal $sig \in Sigs$ is associated with a list $params(sig) = \langle T_{sig,1}, \ldots, T_{sig,k_{sig}} \rangle \in \mathcal{T}^*$ of *parameter types*. The set *Sigs* is partitioned into *system signals SysSigs* and *external signals ExtSigs*. Note that in the level of formal semantics, external signals can have parameters; this is not the case in the actual UML subset due to some technical reasons. A *message* is of form $sig[v_1, \ldots, v_{k_{sig}}]$, where $sig \in Sigs$ and each $v_i \in dom(T_{sig,i})$; the set of all messages is denoted by *Msgs*. Message reception in state machines is specified with *signal triggers* of form $sig(x_1, \ldots, x_{k_{sig}})$, where $sig \in Sigs$ and each x_i is a typed variable with $type(x_i) = T_{sig,i}$. The set of all signal triggers is denoted by *Trigs*.

Classes. A *class* is a pair $C = \langle attrs_C, sm_C \rangle$, where $attrs_C$ is a finite set of typed variables called the *attributes* and sm_C is the *state machine* of the class. Define $Attrs(C) = attrs_C$ and $SM(C) = sm_C$.

3.2 State Machines

The behavior of an instance of a class (i.e. an object) is described by the associated state machine. Formally, a hierarchical UML *state machine* is a structure

$$sm = \langle \mathcal{S}, \mathcal{R}, top, container, \mathcal{T}, defers \rangle,$$

where

- $\mathcal{S}$ is a finite set of *state vertices* partitioned into (i) *simple states* $\mathcal{S}_{si}$, (ii) *composite states* $\mathcal{S}_{co}$, (iii) *final states* $\mathcal{S}_{fi}$, (iv) *initial pseudostates* $\mathcal{S}_{in}$, and (v) *choice pseudostates* $\mathcal{S}_{ch}$;
- $\mathcal{R}$ is a finite set of *regions* (disjoint from $\mathcal{S}$);
- $top \in \mathcal{R}$ is the unique *top region*;
- $container : (\mathcal{S} \cup \mathcal{R} \setminus \{top\}) \to (\mathcal{S} \cup \mathcal{R})$ describes the *state hierarchy* of the state machine;
- $\mathcal{T}$ is a finite set of *transitions*; and
- $defers : (\mathcal{S}_{si} \cup \mathcal{S}_{co}) \to 2^{Sigs}$ assigns each state a (possibly empty) set of *deferrable signals*.

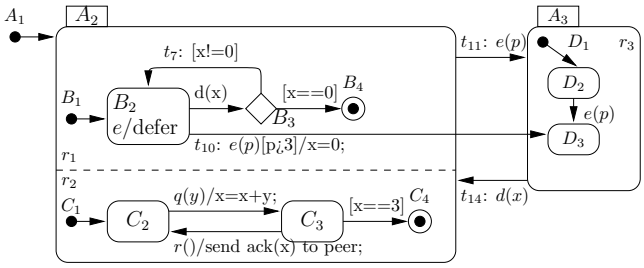

Figure 7: A UML state machine.

For each $v \in \mathcal{S} \cup \mathcal{R}$, define $\mathit{children}(v) = \{v' \in \mathcal{S} \cup \mathcal{R} \setminus \{\mathit{top}\} \mid \mathit{container}(v') = v\}$ and $\mathit{descendants}(v) = \left\{v' \in \mathcal{S} \cup \mathcal{R} \setminus \{\mathit{top}\} \mid \exists i > 0 : \mathit{container}^i(v') = v\right\}$. The state hierarchy must be a connected tree, i.e. $\mathit{descendants}(\mathit{top}) = \mathcal{S} \cup \mathcal{R} \setminus \{\mathit{top}\}$ must hold. It is required that the container of each non-top region is a composite state, i.e. $\forall r \in \mathcal{R} \setminus \{\mathit{top}\} : \mathit{container}(r) \in \mathcal{S}_{co}$ and that the container of each state vertex is a region, i.e. $\forall s \in \mathcal{S} : \mathit{container}(s) \in \mathcal{R}$. Furthermore, each region must contain exactly one initial state, i.e. $\forall r \in \mathcal{R} : |\mathit{children}(r) \cap \mathcal{S}_{in}| = 1$, and each composite state at least one region, i.e. $\forall s \in \mathcal{S}_{co} : \mathit{children}(s) \neq \emptyset$. If a composite state contains more than one region, then it is called *concurrent*. Two state vertices $s_1, s_2 \in \mathcal{S}$ are *orthogonal*, denoted $s_1 \perp s_2$, if there are distinct regions $r_1, r_2 \in \mathcal{R}$ such that $\mathit{container}(r_1) = \mathit{container}(r_2)$, $s_1 \in \mathit{descendants}(r_1)$, and $s_2 \in \mathit{descendants}(r_2)$. A set $S \subseteq \mathcal{S}$ of state vertices is *consistent* iff for any two distinct state vertices $s_1, s_2 \in S$ either $s_1 \perp s_2$, $s_1 \in \mathit{descendants}(s_2)$, or $s_2 \in \mathit{descendants}(s_1)$.

Example 3.1 Consider the state machine in Figure 7. A_2 is a concurrent composite state with $\mathit{container}(A_2) = \mathit{top}$ and $\mathit{children}(A_2) = \{r_1, r_2\}$, where r_1 and r_2 are regions. B_2 is a simple state with $\mathit{defers}(B_2) = \{e\}$ and $\mathit{container}(B_2) = r_1$. The choice pseudostate B_3 and the final state C_4 are orthogonal. The state set $\{A_2, B_1, C_2\}$ is consistent while $\{A_3, D_2, D_3\}$ is not. ♣

A *transition* t in the set $\mathcal{T}$ of transitions is a tuple

$$\langle s, \sigma, g, e, s' \rangle \in (\mathcal{S} \setminus \mathcal{S}_{fi}) \times (\mathit{Trigs} \cup \{\tau\}) \times \mathcal{L}_{\mathbb{B}} \times \mathcal{L}_{\mathrm{Stmt}} \times (\mathcal{S} \setminus \mathcal{S}_{in})$$

with the restriction that if $s \in \mathcal{S}_{in} \cup \mathcal{S}_{ch}$, then $\sigma = \tau$. We define $\mathit{source}(t) = s$, $\mathit{guard}(t) = g$, $\mathit{effect}(t) = e$, $\mathit{target}(t) = s'$, and $\mathit{container}(t) = r$, where $r \in \mathcal{R}$ is the smallest (w.r.t. the partial order induced by *container*) region such that $\{s, s'\} \subseteq \mathit{descendants}(r)$. If $\sigma = \tau$, we say that t is a *completion transition* and define $\mathit{triggersig}(t) = \tau$. Otherwise, $\sigma = \mathit{sig}(\ldots)$ and we define $\mathit{triggersig}(t) = \mathit{sig}$. We require that transitions originating from orthogonal states are not triggered by the same signal: for all $t_1, t_2 \in \mathcal{T}$ it holds that $\mathit{triggersig}(t_1) = \mathit{triggersig}(t_2) \neq \tau$ implies that $\mathit{source}(t_1) \perp \mathit{source}(t_2)$ does not hold.

Example 3.2 In Figure 7, $t_{10} = \langle B_2, e(p), \texttt{p>3}, \texttt{x=0;}, D_3 \rangle$ is a transition with $\mathit{container}(t_{10}) = \mathit{top}$. The completion transition t_7 has $\mathit{container}(t_7) = r_1$, $\mathit{guard}(t_7) = \texttt{x!=0}$ and $\mathit{effect}(t_7) = \texttt{skip}$, where `skip` is a pseudostatement that does nothing. ♣

A *state configuration* of the state machine is a pair

$$sc = \langle A, Q \rangle,$$

where the set A of *active state vertices* is a maximal consistent subset of $\mathcal{S}$ and $Q \subseteq A$ is a set of *quiescent states*. The intuition is that a state is in Q if it has already consumed its implicit "completion event" (that is, the guards of the completion transitions leaving the state have already been evaluated, and will not be evaluated again without re-entering the state). This construction (from [5]) accurately models the use of the implicit completion events in UML without adding them explicitly in the input queue. Because completion events are only relevant for states with outgoing completion transitions, we only define quiescence status for the set of *completion sensitive states* $\mathcal{S}_{CS} = \{s \in \mathcal{S}_{si} \cup \mathcal{S}_{co} \mid \exists t \in \mathcal{T} \text{ s.t. } source(t) = s \text{ and } triggersig(t) = \tau\}$. Thus, Q is always a subset of $A \cap \mathcal{S}_{CS}$. A state $s \in \mathcal{S}_{CS}$ is *ready to consume its completion event in sc*, denoted by $ceready(sc, s)$, if

1. it is active but not quiescent: $s \in A \setminus Q$, and
2. it is either (i) a simple state: $s \in \mathcal{S}_{si}$, or (ii) a composite state with all its regions in final states: $s \in \mathcal{S}_{co}$ and $\forall s' \in \mathcal{S} \cap A : container(container(s')) = s \Rightarrow s' \in \mathcal{S}_{fi}$.

A state configuration $sc = \langle A, Q \rangle$ is

- *in compound transition*, denoted by $inct(sc)$, if it contains an active pseudostate: $A \cap (\mathcal{S}_{in} \cup \mathcal{S}_{ch}) \neq \emptyset$,
- *in run-to-completion (RTC) step*, denoted by $inrtc(sc)$, if (i) it is in compound transition, or (ii) there is a completion sensitive state that is ready to consume its completion event in it, and
- *stable*, denoted by $stable(sc)$, otherwise.

A UML state machine consumes messages from its input queue only when it is in a stable state configuration.

Example 3.3 Consider again the state machine in Figure 7. The state C_3 is the only completion sensitive state in it. The pair $\langle \{A_2, B_2, C_3\}, \{C_3\} \rangle$ is a stable state configuration, while the state configuration $\langle \{A_2, B_3, C_3\}, \{C_3\} \rangle$ is in compound transition. The state configuration $\langle \{A_2, B_2, C_3\}, \emptyset \rangle$ is in RTC step (but not in compound transition) because the state C_3 is ready to consume its completion event in it. ♣

Assume a state vertex $s \in \mathcal{S}$ and a region $r \in \mathcal{R}$ such that $s \in descendants(r)$. The default entry completion of s under r, denoted by $dec(r, s)$, is the smallest maximal consistent subset of $descendants(r) \cap (\mathcal{S}_{co} \cup \mathcal{S}_{in} \cup \{s\})$ such that $s \in dec(r, s)$.

Given a state configuration $sc = \langle A, Q \rangle$ and a transition $t \in \mathcal{T}$ with $source(t) \in A$, the *t-successor* of sc is the state configuration

$$succ\text{-}conf(sc, t) = \langle A', Q' \rangle,$$

where $A' = (A \setminus D) \cup dec(container(t), target(t))$, $Q' = Q \setminus D$, with the abbreviation $D = descendants(container(t))$.

Example 3.4 In the state machine in Figure 7, the t_{10}-successor of the state configuration $sc = \langle\{A_2, B_2, C_3\}, \{C_3\}\rangle$ is $\langle\{A_3, D_3\}, \emptyset\rangle$ while the t_{11}-successor of sc is $\langle\{A_3, D_1\}, \emptyset\rangle$. ♣

3.3 Global Configurations and State Spaces

We next define global configurations, that are snapshots of the global state of the system described by the UML model, and state spaces that consists of all possible global configurations and the information how they may evolve to others.

Global Configurations. We consider a *UML system* to consist of a finite set of objects O, each object $o \in O$ being associated with the class $Class(o)$ that it is an instance of. A *global configuration* of the system is a tuple

$$gc = \langle stateconf_{gc}, attrvals_{gc}, inputq_{gc}, deferq_{gc}\rangle,$$

where

- $stateconf_{gc}$ maps each object o to the current state configuration of its state machine $SM(Class(o))$,
- $attrvals_{gc}$ maps each object o to a function giving each attribute $x \in Attrs(Class(o))$ its current value in $dom(type(x))$, and
- $inputq_{gc}, deferq_{gc} : O \to Msgs^*$ describe the contents of the input and deferred queues, respectively, of each object.

Define $StateConf(gc, o) = stateconf_{gc}(o)$, $AttrVal(gc, o, x) = attrvals_{gc}(o)(x)$, $InputQ(gc, o) = inputq_{gc}(o)$, and $DeferQ(gc, o) = deferq_{gc}(o)$. The set of all global configurations is denoted by GC.

Given a side-effect free Boolean expression ϕ in $\mathcal{L}_{\mathbb{B}}$, $eval(gc, o, \phi)$ evaluates it in the context of a global configuration gc and an object o, and returns *false* or *true*. Given a statement γ in $\mathcal{L}_{\text{Stmt}}$, $exec(gc, o, \gamma)$ executes it in the context of gc and o, and returns a new global configuration gc' with the only restrictions that, for each $o' \in O$, (i) the state machine configuration is not modified: $StateConf(gc', o') = StateConf(gc, o')$, (ii) messages cannot be removed from the input queue: $InputQ(gc, o')$ is a prefix of $InputQ(gc', o')$, and (iii) the deferred queue is not modified: $DeferQ(gc', o') = DeferQ(gc, o')$.

In addition, UML requires that for each pseudostate, there is always at least one outgoing (completion) transition whose guard evaluates to true:

$$\forall gc \in GC, \forall o \in O, \forall s \in \mathcal{S}_{in} \cup \mathcal{S}_{ch}\ \exists t \in \mathcal{T} : \\ source(t) = s \wedge eval(gc, o, guard(t)) = true.$$

State Spaces. The actual semantics of a UML system is given by its *state space* that describes how the system may evolve from one global configuration to another. Each atomic step between global configurations corresponds to one object either firing one transition, deferring a message, or implicitly consuming a message or a completion event. The UML run-to-completion semantics for

individual state machines is followed as messages can only be consumed in stable state configurations. Formally, the *state space* of a UML system is the tuple

$$\langle GC, gc_{\text{init}}, \Delta \rangle,$$

where $gc_{\text{init}} \in GC$ is the *initial configuration*, and $\Delta \subseteq GC \times \mathcal{A} \times GC$ is the minimal *transition relation* defined by the following rules ($\mathcal{A}$ being a set of possible annotations). Assume an object $o \in O$, that $SM(Class(o)) = \langle \mathcal{S}, \mathcal{R}, top, container, \mathcal{T}, defers \rangle$ and let $sc = \langle A, Q \rangle = StateConf(gc, o)$.

- **Signal triggered transitions.**

  If $t = \langle s, sig(x_1, \ldots, x_k), g, e, s' \rangle \in \mathcal{T}$ for a $sig \in SysSigs$, then the *system transition instance* $\langle o, t \rangle$ is *enabled in* gc, denoted by $enabled(gc, \langle o, t \rangle)$, if

  - the state configuration is stable: $stable(sc)$,
  - the source state is active: $s \in A$,
  - the first message in the input queue matches the trigger: $InputQ(gc, o) = \langle sig[v_1, \ldots, v_k], \ldots \rangle$,
  - the guard of the transition evaluates to true: $eval(gc^\star, o, g) = true$, where $gc^\star$ equals to gc except that the message $sig[v_1, \ldots, v_k]$ has been received: $InputQ(gc^\star, o) = dequeue(InputQ(gc, o))$ and $\forall 1 \leq i \leq k : AttrVal(gc^\star, o, x_i) = v_i$,
  - no prioritized transition is enabled: $\nexists t' \in \mathcal{T} : source(t') \in descendants(s) \cap A \wedge enabled(gc, \langle o, t' \rangle)$, and
  - the message is not deferred at a deeper level: $\nexists s'' \in descendants(s) \cap (\mathcal{S}_{si} \cup \mathcal{S}_{co}) \cap A : sig \in defers(s'')$.

  If $enabled(gc, \langle o, t \rangle)$ holds, then

  $$\langle gc, \langle o, t \rangle, gc' \rangle \in \Delta,$$

  where gc' equals to $gc'' = exec(gc^\star, o, e)$ except that (i) $StateConf(gc', o) = succ\text{-}conf(sc, t)$ and (ii) the deferred queue is flushed to the input queue: $InputQ(gc', o) = concat(DeferQ(gc, o), InputQ(gc'', o))$, and $DeferQ(gc', o) = \langle\rangle$.

- **Non-deterministic reception of external events.**

  If $t = \langle s, sig(x_1, \ldots, x_k), g, e, s' \rangle \in \mathcal{T}$ for an external signal $sig \in ExtSigs$, then the *external transition instance* $\langle o, t \rangle$ is $\langle v_1, \ldots, v_n \rangle$*-enabled in* gc, denoted by $enabled(gc, \langle o, t(v_1, \ldots, v_n) \rangle)$, if

  - the state configuration is stable: $stable(sc)$,
  - the source state is active: $s \in A$,
  - the guard of the transition evaluates to true: $eval(gc^\star, o, g) = true$, where $gc^\star$ equals to gc except that the message $sig[v_1, \ldots, v_k]$ has been virtually received: $\forall 1 \leq i \leq k : AttrVal(gc^\star, o, x_i) = v_i$,
  - no prioritized transition with the same signal as trigger is $\langle v_1, \ldots, v_n \rangle$-enabled: $\nexists t' \in \mathcal{T} : triggersig(t') = sig \wedge source(t') \in descendants(s) \cap A \wedge enabled(gc, \langle o, t'(v_1, \ldots, v_n) \rangle)$, and

– the virtual message is not deferred at a deeper level:
$\nexists s'' \in \mathit{descendants}(s) \cap (\mathcal{S}_{si} \cup \mathcal{S}_{co}) \cap A : \mathit{sig} \in \mathit{defers}(s'')$.

If $\mathit{enabled}(gc, \langle o, t(v_1, \ldots, v_n)\rangle)$ holds, then

$$\langle gc, \langle o, t(v_1, \ldots, v_n)\rangle, gc'\rangle \in \Delta,$$

where gc' equals to $gc'' = \mathit{exec}(gc^\star, o, e)$ except that $\mathit{StateConf}(gc', o) = \mathit{succ\text{-}conf}(sc, t)$ and the deferred queue is flushed to the input queue: $\mathit{InputQ}(gc', o) = \mathit{concat}(\mathit{DeferQ}(gc, o), \mathit{InputQ}(gc'', o))$, and $\mathit{DeferQ}(gc', o) = \langle\rangle$.

- **Deferring of messages.**

  If no transition instance is enabled, then the first message in the input queue can be deferred. Formally, the *deferring instance* $\langle o, \text{DEFER}\rangle$ is *enabled* in gc, denoted by $\mathit{enabled}(gc, \langle o, \text{DEFER}\rangle)$, if

  – the state configuration is stable: $\mathit{stable}(sc)$,
  – the input queue is not empty: $\mathit{InputQ}(gc, o) = \langle \mathit{sig}[v_1, \ldots, v_k], \ldots\rangle$,
  – no transition consumes the message:
  $\nexists t \in \mathcal{T} : \mathit{triggersig}(t) = \mathit{sig} \wedge \mathit{enabled}(gc, \langle o, t\rangle)$, and
  – there is an active state deferring the message:
  $\exists s \in (\mathcal{S}_{si} \cup \mathcal{S}_{co}) \cap A : \mathit{sig} \in \mathit{defers}(s)$.

  If $\mathit{enabled}(gc, \langle o, \text{DEFER}\rangle)$ holds, then

  $$\langle gc, \langle o, \text{DEFER}\rangle, gc'\rangle \in \Delta,$$

  where gc' equals to gc except that (i) $\mathit{InputQ}(gc', o) = \mathit{dequeue}(\mathit{InputQ}(gc, o))$ and (ii) $\mathit{DeferQ}(gc', o) = \mathit{append}(\mathit{DeferQ}(gc, o), \mathit{sig}[v_1, \ldots, v_k])$.

- **Implicit consumption of messages.** If the first message in the input queue is not consumed by a transition or deferred, it can be implicitly consumed. Formally, $\mathit{enabled}(gc, \langle o, \text{IMCO}\rangle)$ holds if

  – the state configuration is stable: $\mathit{stable}(sc)$,
  – the input queue is not empty: $\mathit{InputQ}(gc, o) = \langle \mathit{sig}[v_1, \ldots, v_k], \ldots\rangle$,
  – no transition consumes the message:
  $\nexists t \in \mathcal{T} : \mathit{triggersig}(t) = \mathit{sig} \wedge \mathit{enabled}(gc, \langle o, t\rangle)$, and
  – the message is not deferred: $\neg\mathit{enabled}(gc, \langle o, \text{DEFER}\rangle)$.

  If $\mathit{enabled}(gc, \langle o, \text{IMCO}\rangle)$ holds, then

  $$\langle gc, \langle o, \text{IMCO}\rangle, gc'\rangle \in \Delta,$$

  where gc' equals to gc except that $\mathit{InputQ}(gc', o) = \mathit{dequeue}(\mathit{InputQ}(gc, o))$.

- **Firing completion transitions.**

  Completion events are consumed until a stable state configuration is reached. Formally, if $t = \langle s, \sigma, g, e, S'\rangle \in \mathcal{T}$ with $\sigma = \tau$, then the *completion transition instance* $\langle o, t\rangle$ is *enabled in* gc, denoted by $\mathit{enabled}(gc, \langle o, t\rangle)$, if

– the source is active: $s \in A$,

– either (i) the source is a pseudostate: $s \in \mathcal{S}_{in} \cup \mathcal{S}_{ch}$ or (ii) the state configuration is in RTC step but not in compound transition and the source state s ready to consume its completion event: $\neg \mathit{inct}(sc) \wedge \mathit{inrtc}(sc) \wedge s \in \mathcal{S}_{si} \cup \mathcal{S}_{co} \wedge \mathit{ceready}(sc, s)$, and

– the guard condition holds: $\mathit{eval}(gc, o, g) = \mathit{true}$.

If $\mathit{enabled}(gc, \langle o, t \rangle)$ holds, then

$$\langle gc, \langle o, t \rangle, gc' \rangle \in \Delta,$$

where gc' equals to $\mathit{exec}(gc, o, e)$ except that $\mathit{StateConf}(gc', o) = \mathit{succ\text{-}conf}(sc, t)$.

- **Implicit consumption of completion events: quiescing.**

  If a state $s \in \mathcal{S}_{CS}$ is ready to consume its completion event but no outgoing completion transition is enabled, the state can quiesce (i.e. implicitly consume the completion event). Formally, $\mathit{enabled}(gc, \langle o, \text{QUIESCE}_s \rangle)$ holds if

  – the state configuration is in RTC but not in compound transition: $\mathit{inrtc}(sc) \wedge \neg \mathit{inct}(sc)$,

  – the state is active and ready to consume its completion event: $s \in \mathcal{S}_{CS} \cap A \wedge \mathit{ceready}(sc, s)$, and

  – no completion transition leaving s is enabled: $\nexists t \in \mathcal{T} : \mathit{source}(t) = s \wedge \mathit{triggersig}(t) = \tau \wedge \mathit{enabled}(gc, \langle o, t \rangle)$.

  If $\mathit{enabled}(gc, \langle o, \text{QUIESCE}_s \rangle)$ holds, then

  $$\langle gc, \langle o, \text{QUIESCE}_s \rangle, gc' \rangle \in \Delta,$$

  where gc' equals to gc except that $\mathit{StateConf}(gc', o) = \langle A, Q \cup \{s\} \rangle$.

References

[1] Marcus Alanen and Ivan Porres. Coral: A metamodel kernel for transformation engines. In D. H. Akerhurst, editor, *Proceedings of the Second European Workshop on Model Driven Architecture (MDA)*, volume 17-04 of *Technical Report*, pages 165–170. Computing Laboratory, University of Kent, Sep 2004.

[2] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[3] MDE at CREST — the Coral metamodeling toolkit. `http://mde.abo.fi/tools/Coral/`.

[4] Jori Dubrovin. Jumbala — an action language for UML state machines. Research Report A101, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, March 2006.

[5] Toni Jussila, Jori Dubrovin, Tommi Junttila, Timo Latvala, and Ivan Porres. Model checking dynamic and hierarchical UML state machines. In *Proc. MoDeV^{2}a: Model Development, Validation and Verification*, pages 94–110, 2006.

[6] OMG unified modeling language specification version 1.4. Document formal/01-09-67 of the Object Management Group, September 2001.