Towards a Formalization of the UML State Machine using Object-Z

Soon-Kyeong Kim and David Carrington

Department of Computer Science and Electrical Engineering
The University of Queensland, Brisbane, 4072, Australia
Email: soon@csee.uq.edu.au, davec@csee.uq.edu.au

Abstract. This paper presents a formalization using Object-Z of the State Machine package in the UML metamodel. In the specification, the abstract syntax and the static and dynamic semantics for each individual model construct in the package are grouped together and encapsulated within a single Object-Z class. For formalizing the dynamic semantics, a denotational semantics of the construct is first given ignoring detailed operational sequences. Based on this denotational semantics, an operational semantics is then defined in terms of (Object-Z) class operations and invariants constraining the operation sequences. The timed refinement calculus is used to define the operation sequences within Object-Z. Our approach enhances the precision of the UML state machine description and overcomes the lack of modularity, extensibility and reusability of the current UML semantic representation.

1. Introduction

UML is a widely accepted modeling language that can be used to visualize, specify, construct and document the artifacts of a software system [18]. It was accepted as a standard object-oriented modeling language by OMG in 1997, and is becoming the dominant modeling language in industry. The syntax and semantics of notations provided in UML are defined in terms of its metamodel. In the UML metamodel, the modeling constructs are defined using three distinct views: abstract syntax in UML class diagrams, static semantics (ensuring that all UML constructs are statically well formed) in OCL [18], and dynamic semantics (specifying the meaning of the constructs) mainly in English.

As a result of this denotational approach to defining the UML, the abstract syntax, static semantics, and dynamic semantics are defined separately in distinct structures and representations with a lot of redundancy and inconsistencies. Therefore, when a modeling construct is enhanced, each of these three structures also must be extended. The other significant problem of the current UML metamodel is a lack of precision for defining the semantics of modeling constructs. For instance, OCL which is used to specify the static semantics, neither has a precise semantics nor is a consistent formal description language (e.g., OCL flattens set-valued attributes to single-valued attrib-

utes, which makes a consistent logic check over set properties impossible). Also, the English description for defining the dynamic semantics can cause misinterpretations. All these problems limit the precision of UML as a specification language. Several authors [5, 6, 9, 11] have already identified similar deficiencies in UML.

The aim of our work is to develop a complete formal specification of the UML metamodel using Object-Z [3, 21] that can be used with the UML metamodel in a complementary way (refer to [4, 12, 13] for the advantages of using Object-Z for formalizing informal OO models). Previously, we presented an Object-Z specification for the three fundamental packages in the UML metamodel: the Core, DataTypes, and Common Behavior packages [14]. This paper extends our work to the State Machine package.

The State Machine package consists of the modeling concepts used to specify the behavior of various dynamic model elements, which are viewed as statechart diagrams [2, 18]. The state machine in UML is a variation of classical (Harel) statecharts [7] designed to be compatible with concepts in object-orientation. However, there are some semantic differences between the two notations. One major difference is that the *synchrony hypothesis* in the classical statecharts formalism [8, 10, 19, 23] (meaning that transitions take no time to execute and all chain reactions also execute in zero time) is no longer assumed in the UML state machine. In UML, actions may take time and the execution of transitions and events is based on the thread-control mechanism of the software execution model [18] (refer to [18] for other semantic differences relative to classical statecharts). For these reasons, although several semantics have been proposed for classical statecharts [7, 8, 10, 19, 23], it is worthwhile to define a formal semantics for the UML state machine, especially in the context with other UML model elements.

In our work, the abstract syntax and static semantics of the individual model constructs represented as metaclasses in the State Machine package are formalized as attributes and invariants in their corresponding Object-Z classes respectively. The dynamic semantics are formalized in two ways. First, we give denotational semantics for the model constructs ignoring their operational semantics. Based on the denotational semantics given, then we define the operational (execution) semantics of the constructs considering the sequence of operations. Denotational semantics are defined by introducing additional variables (we call these *semantic variables* to distinguish them from the variables that formalize the abstract syntax), which are used to define the semantics and invariants in Object-Z classes. Operational semantics are specified in terms of class operations (we call these operations *metaoperations*) and invariants restricting the operation sequences. We use the timed trace notation of timed refinement calculus [20] to define these invariants with Object-Z. With this approach, we give a mutually consistent (formal) denotational and operational semantics for the UML state machine in a compositional way with respect to its syntax.

We believe our work is the first to integrate the syntax and the denotational and operational semantics of any variant of statecharts into a single model construct such as a class. Latella et al. [16] and Lilius et al. [17] propose a formal definition of part of UML state machines. In our opinion, however, their semantics are not complete and are less formal. Also they do not provide a formal model of the state machine as a

whole. In contrast, our work provides not only a formal definition for individual model concepts but also a formal model for the entire UML state machine package while maintaining the same structure as the UML metamodel in terms of classes. This feature makes an intuitive cross-referencing between the two models possible.

Formalizing and integrating the three distinct views of model constructs into a single construct not only enhances the precision of the UML state machine but also overcomes the lack of modularity, extensibility and reusability of the current UML semantic representation. Moreover, existence of such a formal description for the UML metamodel should make it possible to reason formally about UML models and to formally verify that a system implementation conforms to its UML models as well as providing a precise basis for mapping to other specification languages and for tool-interoperability.

The structure of the rest of this paper is as follows. Section 2 presents the background of our work including a brief description of Object-Z and the timed refinement calculus, and a brief description of how to model UML class constructs using Object-Z class constructs. Section 3 presents a formalization of the abstract syntax and static semantics of model constructs in the State Machine package. Section 4 presents a formalization of the dynamic (execution) semantics of the model constructs. Section 5 concludes our work and describes future work.

2. Background

This section gives a short overview of Object-Z and the timed refinement calculus, and describes how to formalize the three distinct views of UML model constructs using Object-Z classes.

2.1 Object-Z and the timed refinement calculus

Object-Z is an object-oriented extension to Z [22] specifically designed to facilitate specification in an object-oriented style. We summarize features of Object-Z needed in this paper and refer readers to [3, 21] for more details on Object-Z.

Class: In Object-Z, classes are the major modeling construct for specifying a system. A class is a template for objects that have a common state and operations. The state is composed of attributes. Syntactically, a class in Object-Z is a named box, optionally with generic parameters. A class box is composed of a visibility list, type and constant definitions, a state schema, an initial schema and operation schemas. The state schema's declaration part may also be partitioned by a Δ into primary and secondary variables. Secondary variables are used as a convenient means of accessing derivable information (generally from the values of primary variables).

Inheritance and Instantiation: Classes in Object-Z can be used to define other classes by inheritance. A class can inherit from several classes (multiple inheritance). Classes can be instantiated in other classes as attributes. In Object-Z, instantiation is used as a mechanism for modeling relationships between objects, which in UML is modeled using association. Objects which instantiate other classes as their attributes can refer to the objects of the instantiated classes. The values of these attributes are

object-identities of the referenced objects. Object-Z has a reserved word *self*. The value of *self* is the object identity of the current object of the class [3, 21].

Timed refinement calculus: The timed refinement calculus is a Z-based notation for specifying time related properties for real-time systems and it has been integrated with Object-Z for specifying continuous and real-time systems [20]. For brevity, we explain only the subset of the notation needed to specify the operational semantics of the UML state machine (refer to [20] for further details). The notation introduces a given type *T* for absolute time, a function named τ which returns the current time, and an interval bracket $\langle\ \rangle$ to model a set of time intervals. The notation also has a reserved symbol δ denoting the duration of an interval and the concatenation operator ';' for joining intervals whose end points meet.

Integrating the timed refinement calculus in Object-Z extends the structure of Object-Z classes. Classes in the integrated notation are composed of two parts separated by a horizontal line. The part above the line is essentially the standard Object-Z local definitions and schemas (we call this the standard part). The part below the line provides further constraints on the class specified in the timed refinement calculus notation (this is called the timed trace part). All state variables x: X in the standard part are interpreted as timed trace variables x: T $\rightarrow$ X in the timed trace part. The integrated notation [20] also extends the semantics of Object-Z so that operations may take time to execute. Using operation names in the timed trace part denotes the time intervals for executing the operations. The following is an Object-Z class modeling a simple air conditioner.

AirConSystem

$(desiredTemp, INIT)$

$desiredTemp : \mathbb{N}$

$desiredTemp \geq 20$

$temp: \mathbb{N}$
$aircon\_on: Boolean$

TurnOn

$\Delta(aircorn\_on)$

$\neg\ aircon\_on \wedge aircon\_on'$

INIT

$\neg\ aircon\_on$

TurnOff

$\Delta(aircorn\_on)$

$aircon\_on\ \wedge \neg\ aircon\_on'$

SenseTemp

$\Delta(temp)$
$t?: \mathbb{N}$

$temp' = t?$

$\langle\ \exists\ n: \mathbb{N} \bullet \alpha = n^{*}\ 30\ \rangle \subseteq \langle true \rangle;\langle SenseTemp \rangle;\langle true \rangle$
$\langle\ temp > desTemp \wedge \neg\ aircorn\_on \wedge \delta = 10\ \rangle \subseteq \langle true \rangle;\langle TurnOn \rangle;\langle true \rangle$
$\langle\ temp \leq desTemp \wedge aircorn\_on \wedge \delta = 10\ \rangle \subseteq \langle true \rangle;\langle TurnOff \rangle;\langle true \rangle$

In the example, the first invariant in the timed trace part states that for every 30 time units the operation *SenseTemp* occurs to check the current temperature (the symbol α represents the starting point of the interval). The second invariant states that whenever the temperature is greater than the desired temperature, the air conditioner should execute the operation *TurnOn* within 10 time units if it is not on. The LHS is the set of all time intervals where the temperature is greater than the desired temperature for 10 seconds and the system is off. This is a subset of the set of time intervals where the operation *TurnOn* occurs sometime in the interval. Similarly, the third invariant defines when the operation *TurnOff* should occur. In the example, we assume that operations take no time to execute.

2.2 Modeling UML Class Constructs using Object-Z Constructs

In this section, we describe how to model UML class constructs using Object-Z constructs. For brevity, we limit the UML class constructs to those that appear in the State Machine package. Detailed translation rules from UML class constructs to Object-Z constructs are presented in [12] and a formal mapping is given in [13]. Fig. 1 is a fragment of the class diagram taken from the UML semantics [18] showing the state machine in the context of the metaclass ModelElement in the Core package. These metaclasses are used as examples in this section.

Fig. 1. The structure of State Machine with ModelElement

Class: In the UML metamodel, classes are used to show the structure of model elements in terms of metaclasses. We model each UML metaclass using a separate Object-Z class. When attributes appear in a metaclass, the attributes and their types are used as attributes and attribute types in its Object-Z class.

Association: An association is modeled as an attribute of the corresponding Object-Z classes that it relates. For each associated class, the type of the attribute is the corresponding Object-Z class of its opposite class. For composition, the symbol © which represents unshared ownership in Object-Z [3] is added to the attribute name. When an association is directed (shown by an arrow at one association end), only objects of the class to which the arrow is not attached can access objects of the class to which the arrow is attached. In this case, the association is modeled only as an attribute of the Object-Z class that can access. Multiplicity constraints are formalized as invariants in the predicate of Object-Z classes.

Inheritance: In the UML metamodel, inheritance is used to show taxonomic and hierarchical relationships between model elements. We model these taxonomic relationships between model elements using the inheritance mechanism in Object-Z. For each metaclass that inherits from other metaclasses, the names of the Object-Z classes corresponding to their superclasses are included within the Object-Z class corresponding to the metaclass. Since in UML, generalization is a subtyping relationship, a down-arrow symbol (denoting subtyping in Object-Z) is added whenever superclasses are used as a type within other classes.

The following Object-Z class is a formal description of ModelElement in Fig. 1. The association to StateMachine is modeled as an attribute, with the symbol Ⓢ representing shared ownership in Object-Z [1]. A constraint formalizing the links between the instances of the two metaclasses are added in the predicate part (also see the Object-Z class *StateMachine* in Section 3.6).

ModelElement

$\ldots$

$\ldots$

$behavior : \mathbb{P}\ StateMachine\,Ⓢ$

$\forall s: behavior \bullet self \in s.context$

3. Formalization of the abstract syntax and static semantics of model elements in the State Machine Package

In this section, we first formalize the abstract syntax and static semantics of individual model constructs in the State Machine package. The state machine as a whole is then defined by composing its components. Fig. 2 is part of a class diagram showing the model elements in the State Machine package [18]. Since space does not allow us to present all model elements in the state machine package, we present only core elements and refer to [15] for the complete description. In the Object-Z classes defined in this paper, for brevity we omit some constraints that formalize links between associated classes. We also assume that types Name, Classifier and BehavioralFeature are defined as Object-Z classes (see [14]).

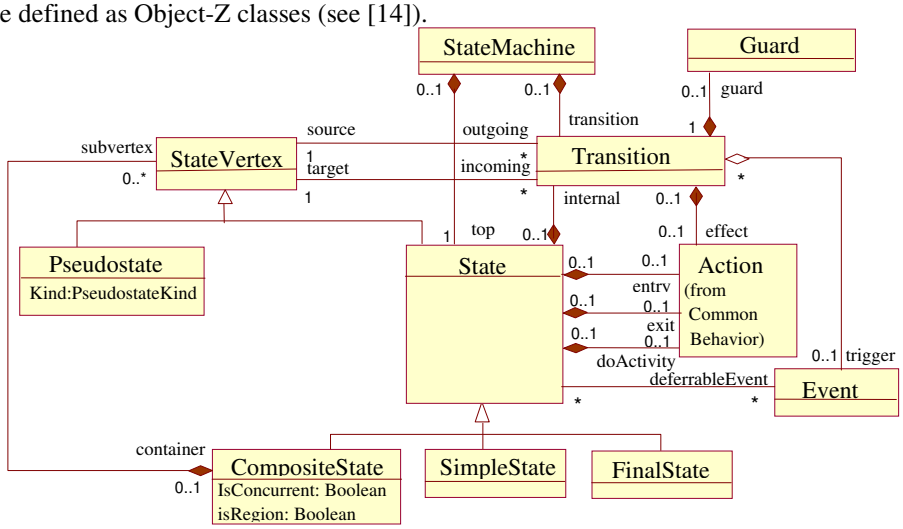


Fig. 2. State Machine -Main

3.1 State

Syntactically, State inherits from StateVertex and has several associations with StateMachine, Action, Event and Transition: *stateMachine*, *entry*, *doActivity*, *exit*, *deferrableEvent*, and *internal*. The associations are formalized as attributes in the Object-Z class *State* below. We assume that StateVertex is also formalized as an Object-Z class. No specific static semantics are given for states.

State

StateVertex

$$
\begin{array}{l}
stateMachine : \mathbb{P}\ StateMachine \\
entry : \mathbb{P}\downarrow\ Action\ \copyright \\
doActivity : \mathbb{P}\downarrow\ Action\ \copyright \\
exit : \mathbb{P}\downarrow\ Action\ \copyright \\
deferrableEvent : \mathbb{P}\downarrow\ Event \\
internal : \mathbb{P}\ Transition\ \copyright \\
\hline
\{\#\ stateMachine, \#\ entry, \#\ exit, \#\ doActivity\} \subseteq \{0, 1\}\ \ldots
\end{array}
$$

3.2 Pseudostate

The kinds of pseudo states in UML are: *initial*, *deepHistory*, *shallowHistory*, *join*, *fork*, *junction*, and *choice*. For brevity, we omit detailed structure of pseudo states.

$PseudostateKind := initial \mid deepHistory \mid shallowHistory \mid join \mid fork \mid junction \mid choice$

Pseudostate

StateVertex

$kind : PseudostateKind$

$\ldots$

3.3 CompositeState

CompositeState has two attributes *isConcurrent* and *isRegion* representing whether it is decomposed into two or more orthogonal regions and whether it is a substate of a concurrent state respectively [18]. It inherits from State and has a composition association to StateVertex. The static semantics given for composite states are:

[1] A composite state can have at most one initial vertex.
[2] A composite state can have at most one deep history vertex.
[3] A composite state can have at most one shallow history vertex.
[4] There must be at least two composite substates in a concurrent composite state.
[5] A concurrent state can only have composite states as substates.
[6] The substates of a composite state are part of only that composite state. Note that this rule is already defined by the composition relationship between the metaclass StateVertex and CompositeState (the symbol © added to the attribute *subvertex* formalizes this rule implicitly).

The static semantics are formalized as invariants in the predicate (note that numbers are purely descriptive).

CompositeState

State

$isConcurrent : Boolean$
$isRegion : Boolean$
$subvertex : \mathbb{P} \downarrow StateVertex$ ©

$[1]\ \#\{\ s\colon subvertex \mid s \in Pseudostate \wedge s.kind = initial\ \} \leq 1$
$[2]\ \#\{\ s\colon subvertex \mid s \in Pseudostate \wedge s.kind = deepHistory\ \} \leq 1$
$[3]\ \#\{\ s\colon subvertex \mid s \in Pseudostate \wedge s.kind = shallowHistory\ \} \leq 1$
$[4]\ isConcurrent \Rightarrow \#\{\ s\colon subvertex \mid s \in CompositeState\ \} \geq 2$
$[5]\ isConcurrent \Rightarrow \forall\ s\colon subvertex \bullet s \in CompositeState$

3.4 Event and Actions

Event and Action inherit from ModelElement. Event has parameters and associations to Transition and State. The four kinds of events provided in UML are: *SignalEvent, CallEvent, TimeEvent*, and *ChangeEvent.* Action has an attribute *isAsynchronous* indicating whether or not the action is asynchronous and several other attributes (refer to [14] for details of actions). Two major kinds of actions defined in UML are *Cal-*

lAction which results in invoking an operation on an instance synchronously and *SendAction* which results in sending a (asynchronous) signal. In this paper, we discuss only part of the semantics of actions and events. The following Object-Z classes are formal descriptions of events and actions.

Event

ModelElement

$transition : \mathbb{P}\ Transition$
$state : \mathbb{P}\downarrow State$
$parameters : seq\ Parameter\ ©$
...

Action

ModelElement

$isAsynchronous : Boolean$
$entryState : \mathbb{P}\ State$
$exitState : \mathbb{P}\ State$
$activityState : \mathbb{P}\ State$
...

3.5 Transition

Transition has a source and a target state vertex. It also can have a guard, which provides a condition for firing the transition. The static semantics of transitions are:

[1] Transitions outgoing from pseudo states may not have a trigger. Note that this rule is inconsistent with rule [8] below, so we restrict the rule to non-initial transitions.

[2] A fork segment should not have guards or triggers. Note that part of the semantics related to triggers is already defined in the rule [1] above so it is redundant.

[3] A join segment should not have guards or triggers.

[4] A fork segment should always target a state.

[5] A join segment should always originate from a state.

[6] Join segments should originate from orthogonal states.

[7] Fork segments should target orthogonal states.

[8] 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 the BehavioralFeature. Apart from these cases, an initial transition never has a trigger.

A formal description of transitions is given below.

Transition

ModelElement

$stateMachine : \mathbb{P}\ StateMachine$
$guard : \mathbb{P}\ Guard\ ©$
$source : \downarrow StateVertex$
$target : \downarrow StateVertex$
$trigger : \mathbb{P}\downarrow Event$
$state : \mathbb{P}\downarrow State$
$effect : \mathbb{P}\downarrow Action\ ©$

$\{\#\ stateMachine, \#\ guard, \#trigger, \#state, \#effect\} \subseteq \{0, 1\} \ldots$
$[1]\ source \in Pseudostate \land source.kind \neq initial \Rightarrow trigger = \varnothing$
$[2]\ source \in Pseudostate \land source.kind = fork \Rightarrow guard = \varnothing$
$[3]\ target \in Pseudostate \land target.kind = join \Rightarrow guard = \varnothing \land trigger = \varnothing$
$[4]\ stateMachine \neq \varnothing \land source \in Pseudostate \land source.kind = fork \Rightarrow$
$target \in State$
$[5]\ stateMachine \neq \varnothing \land target \in Pseudostate \land target.kind = join \Rightarrow$
$source \in State$
$[6]\ target \in Pseudostate \land target.kind = join \Rightarrow$
$\forall\ c : source.container \bullet c.isConcurrent$

$$
\begin{array}{l}
[7]\ source \in Pseudostate \wedge source.kind = fork \Rightarrow \\
\qquad\qquad \forall\, c: target.container \bullet c.isConcurrent \\
[8]\ source \in Pseudostate \wedge source.kind = initial \Rightarrow\ trigger = \varnothing \vee \\
(\ source.container = \{s: stateMachine \bullet s.top\} \wedge \\
\qquad \cup\{s: \cup\{t: trigger \bullet t.stereotype\} \bullet s.name\}) \subseteq \{create\}) \vee \\
(\ \cup\{s: stateMachine \bullet s.context\} \subseteq BehavioralFeature \wedge trigger \subseteq CallEvent \wedge \\
\qquad \{t: trigger \bullet t.operation\} = \cup\{s{:}stateMachine \bullet s.context\})
\end{array}
$$

3.6 StateMachine

StateMachine inherits from ModelElement and has composition relationships to State (representing the top state) and Transition (representing a set of transitions that belong to the state machine). It also has an aggregation relationship to ModelElement which is the context of the state machine. The static semantics are:

[1] A StateMachine is aggregated within either a classifier or a behavioral feature.

[2] A top state is always a composite. Note that this rule can be removed by modifying the association to State as an association to CompositeState instead.

[3] A top state cannot have any containing states.

[4] The top state cannot be the source of a transition.

[5] If a StateMachine describes a behavioral feature, it contains no triggers of type CallEvent, apart from the trigger on the initial transition (also see the static semantics [8] in the Object-Z class *Transition*).

A formal description of State is given below.

StateMachine

ModelElement

$$
\begin{array}{l}
context : \mathbb{P}\downarrow ModelElement \\
top : \downarrow State\ \copyright \\
transition : \mathbb{P}\ Transition\ \copyright \\
\ldots \\
\hline
\#\, context \leq 1 \wedge \forall c : context \bullet self \in c.behaviour \wedge \{self\} \in top.stateMachine \\
\forall t : transition \bullet self \in t.stateMachine\ \ldots \\
[1]\ \forall c : context \bullet c \in Classifier \vee c \in BehavioralFeature \\
[2]\ top \in CompositeState \\
[3]\ top.container = \varnothing \\
[4]\ top.outgoing = \varnothing \\
[5]\ \forall\, c: context \bullet c \in BehavioralFeature \Rightarrow \\
\qquad \{t: transition \mid \neg\ (\ t.source \in Pseudostate \wedge t.source.kind = initial)\ \wedge \\
\qquad\qquad t.trigger \subseteq CallEvent\ \} = \varnothing
\end{array}
$$

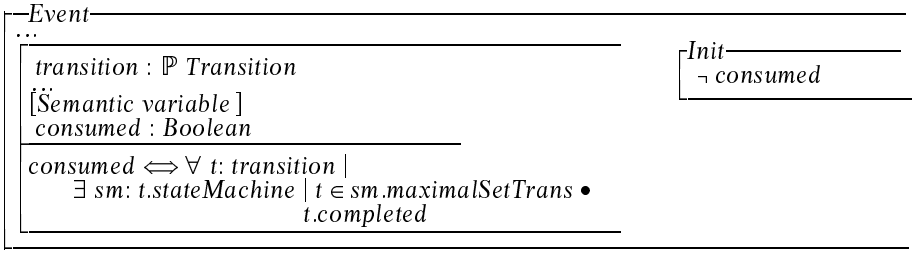
4. Formalization of the dynamic semantics of model elements in the State Machine Package

In the UML State Machine package, the semantics of state machines are specified in terms of the operations of a hypothetical machine that implements a state machine specification [18]. The hypothetical machine is composed of an event queue holding incoming events, an event dispatcher mechanism and an event processor which processes dispatched event instances according to the semantics of the state machine.

UML does not provide any specific requirements for the event queue and dispatcher mechanism leaving open the possibility of modeling different schemes [18]. For this reason, we do not provide a formal definition of the event queue and dispatcher mechanism concentrating on the event processor which refers to the state machine in the metamodel.

4.1 Event

An event instance is a trigger of a transition. When an event instance arrives at the state machine, if the state machine is not blocked with another event, the event is referred to as the *current event*. When the event processing is completed, the event is *consumed*. To formalize these semantics, we define a (semantic) variable *consumed* of type Boolean. The variable is false until the event is completely processed by the state machine. When an event triggers more than one transition, the transitions can conflict with each other. In this case transitions to be fired are selected by the transition selection algorithm discussed in Section 4.5. Consequently, an event is said to be fully *consumed* when all transitions enabled by the event that are also selected to fire by the state machine are completed. This property is defined as an invariant in the Object-Z class *Event* (also see the Object-Z class *Transition* and *StateMachine*).

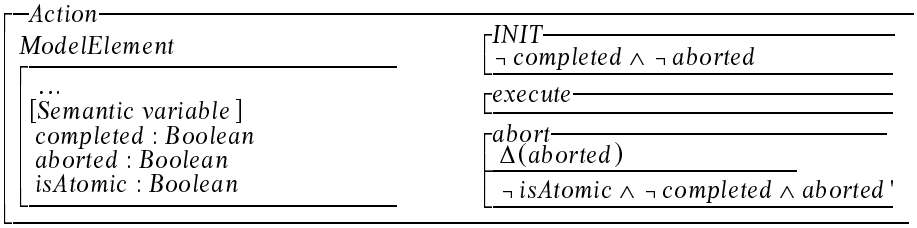


4.2 Action

An action is a specification of an executable statement that should be performed as a result of the execution of a transition or entering to or exiting from a state. We extend the Object-Z class *Action* as follows. The Object-Z class *Action* now has a semantic variable *completed* of type Boolean representing its state. For synchronous actions, this variable becomes true when the execution of the actions is completed. For asynchronous actions, the variable becomes true when the actions are executed, but without waiting for the completion of the actions. We also define an abstract operation *execute* to represent the necessary realization of the action. This operation is invoked by other model elements such as states and transitions, which need to execute the action in the context of their behaviors. Since the actual realization of the execution of an action is more complex and it should be described in the context with other dynamic model elements in UML such as Stimulus, Collaboration, Message, and so on, we do not further define the execution semantics of actions in this paper.

Note that UML states that the execution of actions is atomic, that is non-interruptible. In contrast, the execution of activities is interruptible, so they can be aborted. Since activities are also actions in UML, this execution semantics of activi-

ties is in conflict with the execution semantics of actions. To overcome this semantic conflict, we extend *Action* by adding an additional semantic variable, *isAtomic* of type Boolean. This attribute is true when the instances of Action model atomic actions, while it is false when the instances of Action model interruptible actions, which are defined as activities in UML. We also define a semantic variable, *aborted* of type Boolean representing whether or not the action has been aborted. Based on this semantic extension, we now define an additional operation, *abort*, which can cause an interrupt on the execution of the action. This operation only can occur when the value of the attribute *isAtomic* is false and it makes the attribute *aborted* true as the result of the operation.



4.3 State

The dynamic semantics of states are formalized in two ways: denotational and operational. We formalize the following semantics of states:

- In a hierarchical state machine, if a state is *active*, all of its containing states should also be active.
- A state is said to be *completed* when its entry action, transition and activity are completed. In this case a completion event is generated implicitly (see StateMachine in Section 4.5 for the generation and processing of completion events).

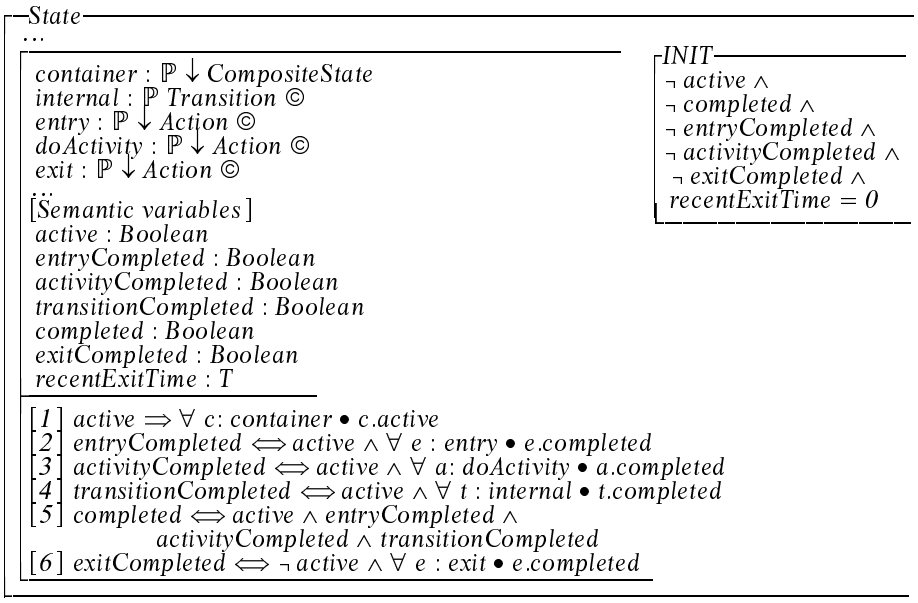
We also formalize the following execution sequence for states.

- When a state is entered, the following execution sequence should occur.
  1. The state becomes active and the entry action is executed prior to any other action. When a state is contained by a composite state, the entry action of the state is executed after the entry action of the composite state is executed.
  2. When the entry action is completed, activity defined in the state is executed.
- In contrast, when a state is exited, the following execution sequence should occur.
  1. If the activity is incomplete, the activity is aborted.
  2. The exit action is executed.

Formalizing the denotational semantics: We first define several semantic variables and then formalize the denotational semantics of states in terms of invariants using these definitions. The semantic variables are:

- *active* of type Boolean which becomes true when the state is entered and becomes false when the state is exited (the invariant [1] defined in the predicate states that when a state is active, all enclosing states should also be active);

- *entryCompleted*, *activityCompleted*, *transitionCompleted*, and *completed* of type Boolean; each of them represents the completion of the behavior of its corresponding model element (constraints specifying when these variables become true are defined as invariants, [2] to [5] below);
- *exitCompleted* which represents the state of the exit action (see invariant [6])
- a (time) variable *recentExitTime* which records the time when the state is the most recently exited. This time variable is used to determine history states.



Formalizing the operational semantics: Prior to formalizing the execution semantics for states, we define several operations:

- The operation *enter* changes the variable *active* from false to true.
- The operation *exit* changes the variable *active* from true to false and records the exiting time.
- The operation *OuterEnter* applies the operation itself to the all containing states recursively if they have not been entered yet and enters the state by invoking the operation *enter*.
- The operations *ExecuteEntry, ExecuteActivity* and *ExecuteExit* execute the entry action, the activity and the exit action respectively by invoking the operation *execute* defined in the Object-Z class *Action*.
- The operation *AbortActivity* results in aborting the activity (if it is not completed) by invoking the operation *abort* defined in the Object-Z class *Action*.

We also define two operations *Enter* and *Exit* which are visible and are used by transitions when they enter or exit a state (see the Object-Z class *Transition* in Section 4.4). Since entering a state should start from the outermost ancestor state if it has not been entered, the operation *Enter* invokes the operation *OuterEnter*. This guarantees that the containing states of a state are always active when the state is active. The *Exit* operation simply exits the state by invoking the operation *exit*.

State

$\uparrow(Enter, Exit)$

enter

$\Delta(active)$

$\neg\, active \wedge \neg\, completed \wedge active'$

exit

$\Delta(active, completed, recentExitTime)$

$active \wedge \neg\, active' \wedge \neg\, completed'$
$recentExitTime' = \tau$

$OuterEnter \triangleq (\bigwedge c: container \mid \neg\, c.active \bullet c.OuterEnter) \wedge enter$
$ExecuteEntry \triangleq \bigwedge e: entry \bullet e.execute$
$ExecuteActivity \triangleq \bigwedge a: doActivity \bullet a.execute$
$AbortActivity \triangleq \bigwedge a: doActivity \mid \neg\, a.completed \bullet a.abort$
$ExecuteExit \triangleq \bigwedge x: exit \bullet x.execute$
$Enter \triangleq OuterEnter$
$Exit \triangleq exit$

Finally, we formalize the operation sequences of a state in terms of invariants defined in the timed trace part. Prior to describing the invariants, we explain that how the invariants are interpreted with an example. $\langle \neg P\rangle;\langle P\rangle \subseteq \langle \neg OP\rangle;(\langle OP\rangle \cup \langle OP\rangle;\langle true\rangle)$ means that the operation *OP* occurs immediately when the system in which the example is defined enters an interval where *P* is true following an interval where *P* is not true. The part of the RHS $(\cup \langle OP\rangle;\langle true\rangle)$ ensures that the execution of the operation finishes during the interval where *P* is true. An informal description of the invariants defined in the timed-trace part is given as follows:

- Invariant [1] formalizes the operation sequence when a state is entered. The invariant ensures that the operation *ExecuteEntry* occurs immediately when the state is active and the entry action of its container state is executed.
- Invariant [2] formalizes the operation sequence when the entry action is completed. It states that when both the state is active and the entry action is completed, the activity is executed immediately by invoking the operation *ExecuteActivity*.
- Invariant [3] formalizes the operation sequence when a state is exited (inactive), but the activity is not completed. In this case, the activity is aborted by invoking the operation *AbortActivity*.
- Invariant [4] formalizes the operation sequence when a state is exited. The invariant ensures that the operation *ExecuteExit* occurs when both the state is not active (exited) and the activity has been completed or aborted. Since the execution sequence for exiting from composite states is slightly different, the invariant is restricted to non-composite states.

State

…

$[1]\ \langle \neg\, (active \wedge (\forall\, c: container \bullet c.entry.completed))\rangle;$
$\langle active \wedge (\forall\, c: container \bullet c.entry.completed)\rangle \subseteq$
$\langle \neg\, ExecuteEntry\rangle;(\langle ExecuteEntry\rangle \cup \langle ExecuteEntry\rangle;\langle true\rangle)$

$[2]\ \langle active \wedge \neg\, entryCompleted\rangle;\langle active \wedge entryCompleted\rangle \subseteq$
$\langle \neg\, ExecuteActivity\rangle;(\langle ExecuteActivity\rangle \cup \langle ExecuteActivity\rangle;\langle true\rangle)$

$[3]\ \langle active \wedge \neg\, activityCompleted\rangle;\langle \neg\, active \wedge \neg\, activityCompleted\rangle \subseteq$
$\langle \neg\, AbortActivity\rangle;(\langle AbortActivity\rangle \cup \langle AbortActivity\rangle;\langle true\rangle)$

$[4]\ \forall\, s: \downarrow State \mid s \notin CompositeState \bullet$
$\langle active \vee \neg\, (\forall\, a: doActivity \bullet a.completed \vee a.aborted)\rangle;$
$\langle \neg\, active \wedge (\forall\, a: doActivity \bullet a.completed \vee a.aborted)\rangle \subseteq$
$\langle \neg\, ExecuteExit\rangle;(\langle ExecuteExit\rangle \cup \langle ExecuteExit\rangle;\langle true\rangle)$

Formalizing composite states: A composite state is composed of a set of states which may have their own state hierarchies. UML refers to these state hierarchies as *state configurations*. Space does not allow us to present details, so we give only part of the Object-Z class formalizing composite states below (refer to [15] for details).

CompositeState

$$
\begin{array}{|l}
\hline
\ldots \\
\Delta \\
stateConfiguration : \mathbb{P}\downarrow\ StateVertex \\
\hline
\ldots \\
stateConfiguration = subvertex \cup (\cup \{\ s{:}\ subvertex \mid s \in CompositeState \bullet \\
\qquad\qquad s.stateConfiguration\}) \\
\hline
\end{array}
$$

4.4 Transition

The major semantics of transitions are the enabling mechanism and execution sequences of transitions. A transition is enabled if and only if [18]:

- All of its source states are in the active state configuration of the state machine.
- The trigger of the transition is satisfied by the current event.
- If a guard exists, it should be true.
- There exist at least one full path from the source state to the target state. This means that there should exist a common ancestor containing the source and target states in its state configuration.

Being enabled does not guarantee that the transition is selected to fire (the transition selection mechanism is discussed with the state machine in the following section). Once a transition is enabled and is selected to fire, the transition is said to be *completed* when all the necessary execution steps described in the operational semantics are completed. The least common ancestor is the state that contains all the explicit source states and explicit target states of a transition. The main source state is a direct substate of the least common ancestor that contains the explicit source states. The main target state is a direct substate of the least common ancestor that contains the explicit target states.

Formalizing the denotational semantics: We define several secondary variables formalizing the notions introduced above: *leastComAncestor* representing the least common ancestor of a transition, *mainSource* representing the main source of a transition, *mainTarget* representing the main target of a transition, *explicitSourceStates* representing all the explicit source states, and *explicitTargetStates* representing all the explicit target states. For brevity, we omit details of the variables (refer to [15]).

Transition

$$
\begin{array}{|l}
\hline
\ldots \\
\Delta \\
leastComAncestor : \downarrow CompositeState \\
mainSource : \downarrow StateVertex \\
mainTarget : \downarrow StateVertex \\
explicitSourceStates : \mathbb{P}\downarrow StateVertex \\
explictTargetStates : \mathbb{P}\downarrow StateVertex \\
\ldots \\
\hline
\end{array}
$$

We also define several semantic variables: *enabled, effectCompleted, exitCompleted, entryCompleted* and *completed* of type Boolean. The variable *enabled* becomes true when the transition satisfies all the conditions for enabling transitions described above (see invariant [1]). Transitions that do not have triggers (such as transitions entering to or originating from join pseudo states) are not enabled explicitly by the rules described above (completion transitions are not included in this scheme because they have implicit triggers, completion events). Instead, these transitions take place immediately when their source states become active. For this reason, we limit transitions discussed in this section to those that have explicit or implicit triggers (see the constraint $trigger \neq \emptyset$ in invariant [1]). Note that the *currentEvent* is defined in the Object-Z class *StateMachine* in Section 4.5 representing the current event of the state machine.

The variables *effectCompleted, exitCompleted* and *entryCompleted* become true when the actions on the transition (effect), the exit actions of all the explicit source states, and the entry action of all the explicit target states are completed respectively (see invariant [2], [3] and [4]). Finally, the variable *completed* becomes true when the three variables *entryCompleted*, *effectCompleted* and *exitCompleted* become true (see invariant [5]). For brevity, we do not present the execution semantics for entering pseudo states. In this sense, the Object-Z specification represented in this paper is not complete. For the complete formal model, refer to [15].

Transition

…

$stateMachine : \mathbb{P}\ StateMachine\ ©$
$guard : \mathbb{P}\ Guard\ ©$
$source : \downarrow StateVertex$
$target : \downarrow StateVertex$
$trigger : \mathbb{P} \downarrow Event$
$effect : \mathbb{P} \downarrow Action\ ©$

…

[*Semantic variables*]
$enabled : Boolean$
$exitCompleted : Boolean$
$effectCompleted : Boolean$
$entryCompleted : Boolean$
$completed : Boolean$

INIT

$\neg\ enabled \wedge$
$\neg\ completed \wedge$
$\neg\ exitCompleted \wedge$
$\neg\ effectCompleted \wedge$
$\neg\ entryCompleted \wedge$

[1] $enabled \Leftrightarrow \forall\ s: explicitSourceStates \bullet s.active \wedge$
$\forall\ g: guard \bullet g.expression \wedge$
$trigger \neq \emptyset \wedge trigger = \cup\{s: stateMachine \bullet s.currentEvent\} \wedge$
$\exists_1\ s: \downarrow CompositeState \bullet \{source, target\} \subseteq s.stateConfiguration$
[2] $effectCompleted \Leftrightarrow enabled \wedge \forall\ e: effect \bullet e.completed$
[3] $exitCompleted \Leftrightarrow enabled \wedge$
$\forall\ x: explicitSourceStates \mid x \in \downarrow State \bullet x.exitCompleted$
[4] $entryCompleted \Leftrightarrow enabled \wedge$
$\forall\ e: explicitTargetStates \mid e \in \downarrow State \bullet e.entryCompleted$
[5] $completed \Leftrightarrow exitCompleted \wedge effectCompleted \wedge entryCompleted$

Operational Semantics: When a transition is enabled and is selected to fire, the following steps are carried out in order [18].

- The main source state is properly exited, which means that all the exit actions of the explicit source states have been executed (completed).

- Actions on the transition are executed. If actions are synchronous, the transition is not completed until the invoked objects complete all the synchronous actions (for synchronous actions, the semantic variable *completed* defined in the Object-Z class *Action* cannot be true until the actions are completed).
- The target state is properly entered, which means that all the entry actions in the explicit target states are executed.

Formalizing the operational semantics: To formalize the operational semantics, we define several operations: *Fire*, *ExitSource*, *ExecuteEffect*, and *EnterTarget*.

- The operation *Fire* is invoked by the state machine. The precondition of the operation is that the transition should be enabled. Invoking this operation causes execution of the exit actions which is followed by executing the operation *ExitSource*.
- The operations *ExitSource* and *EnterTarget* invoke the execution of the exit actions invoking the *Exit* operation in the main source state and the execution of the entry actions invoking the *Enter* operation in the target state respectively. The rest of the exit execution step (or the entry execution) is carried out by the *Exit* operation (or the *Enter* operation) recursively.
- The operation *ExecuteEffect* invokes the execution of all actions defined on the transition.

Transition

$$
\begin{aligned}
&\ldots \\
&Fire \triangleq [\, enabled \,] \wedge ExitSource \\
&ExitSource \triangleq [\neg\, completed \,] \wedge mainSource.Exit \\
&ExecuteEffect \triangleq [\neg\, completed \,] \wedge (\bigwedge e: effect \bullet e.execute) \\
&EnterTarget \triangleq [\neg\, completed \,] \wedge target.Enter
\end{aligned}
$$

Finally, we formalize the execution sequence of a transition using these operations in terms of invariants [1] and [2] in the timed trace part.

- The invariant [1] states that when the exit execution is completed, the *ExecuteEffect* operation should occur immediately, which results in executing actions on the transition.
- The invariant [2] states that when all the effect actions are completed, the operation *EnterTarget* should occur immediately, which results in executing all the entry actions.

Transition

$$
\begin{aligned}
&\ldots \\
&[1]\ \langle \neg\, exitCompleted \rangle ; \langle exitCompleted \rangle \subseteq \\
&\qquad \langle \neg\, ExecuteEffect \rangle ; (\langle ExecuteEffect \rangle \cup \langle ExecuteEffect \rangle ; \langle true \rangle) \\
&[2]\ \langle \neg\, effectCompleted \rangle ; \langle effectCompleted \rangle \subseteq \\
&\qquad \langle \neg\, EnterTarget \rangle ; (\langle EnterTarget \rangle \cup \langle EnterTarget \rangle ; \langle true \rangle)
\end{aligned}
$$

4.5 StateMachine

In the UML metamodel, the semantics of state machines are described in terms of the run-to-completion assumption and transition firing mechanisms.

Run-to-completion assumption: Under this assumption, an event can only be dequeued and dispatched for processing when the processing of the previous current event is fully completed [18]. UML applies this assumption to the entire state ma-

chine including the concurrent steps taken by concurrent regions in the active state configuration of the state machine. When all transitions (in concurrent composite states or not) fired by the current event complete their steps as described in section 4.4, the current event is fully *consumed* and the run-to-completion step is *completed.* Since the execution of a transition is completed when all the exit, effect and entry actions caused by the transition are completed (see section 4.4), when a run-to-completion step is finished, the state machine is in a stable state configuration with all actions (but not necessarily activities) completed.

Transition firing mechanism: Events can fire more than one transition. In this case, the transitions may conflict with one another. Two possible cases are [18]:

- When two transitions are triggered by the same event originating from the same source state, but with different guards. In this case, only one of them whose guard is true will fire in a single run-to-completion step.
- When the intersection of the set of states (the explicit source states) that two transitions exit is non-empty.

Maximal set transitions: UML provides transition selection rules that exclude all conflicting transitions from the set of transitions which will fire. This set is called the *maximal set transitions* [18]. Transitions included in the maximal set transitions should satisfy the following properties:

- All transitions are enabled.
- There are no conflicting transitions in the set.
- There is no transition outside of the set that has higher priority than a transition in the set. The firing priority is that a transition originating from a substate has higher priority than a conflicting transition originating from any of its containing states [18]. However, if the source states of two conflicting transitions have the same state configuration, no priority is given.

Formalizing the denotational semantics: Prior to formalizing the semantics, we define two secondary variables that formalize the notions introduced above: *maximalSetTrans* representing all transitions that will fire and *activeStateConfiguration* representing all active states in the state configuration. Due to the space limitation, we omit detailed definitions of these variables.

StateMachine

$$\ldots$$

$$\Delta$$

$$maximalSetTrans : \mathbb{P}\ Transition$$

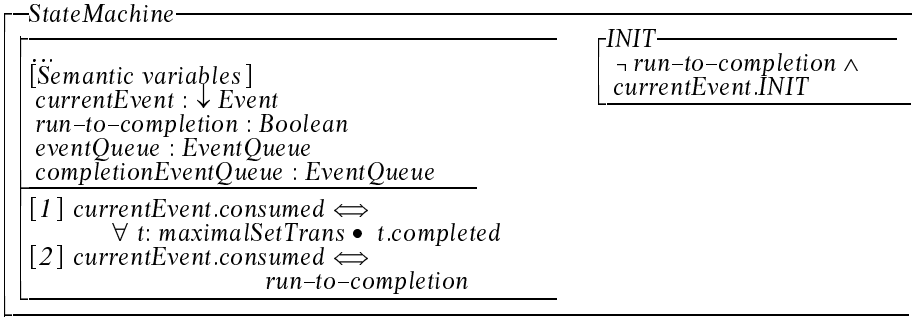
$$activeStateConfiguration : \mathbb{P}\ \downarrow StateVertex$$

$$\ldots$$

We also define the semantic variables *currentEvent* of type Event representing the current event in the state machine and *run-to-completion* of type Boolean representing the status of the run-to-completion step. Since a state machine dispatches and processes the dispatched events, we define an abstract Object-Z class say *EventQueue* without defining details (because UML does not give any semantics with respect to the event queue and the event dispatcher mechanism). The class has two operations *dequeue* (which delivers a new event to the state machine according to the event dispatcher mechanism) and *enqueue* (which adds a new event into the queue). The

class also has an operation *empty* returning a Boolean value denoting whether or not the queue is empty. In the Object-Z class *StateMachine*, we define two event queues: *eventQueue* and *completionEventQueue*. The variable *eventQueue* represents an event queue for external events. In contrast, the variable *completionEventQueue* represents an event queue for completion events that are generated when all transition, entry actions and activities in the currently active state are completed.

The invariant [1] and [2] formalize the run-to-completion assumption. Invariant [1] defines the condition for the current event to be consumed. This is, when all the transitions in the maximal set transition are completed, the current event is consumed. Invariant [2] formalizes that when the current event is consumed, the run-to-completion is completed.

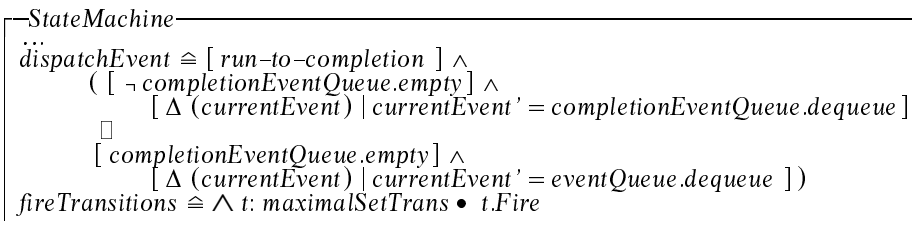


The operational semantics: The operational semantics of the state machine are formalized in the timed trace part:

[1] When the run-to-completion step is completed, the state machine dispatches a new event by invoking the operation *dispatchEvent* which in turn invokes the operation *dequeue* defined in the event queue. If the *completionEventQueue* is not empty (meaning that there exists a completion event generated), the operation *dispatchEvent* dispatches an event from the *completionEventQueue*. Otherwise, the operation dispatches a new event from the *eventQueue*. We treat completion transitions in the same way as other transitions, which can be enabled by a completion event.

[2] When a new event is delivered, the state machine processes the event by invoking the operation *fireTransitions*, which fires all states in the maximal set transitions by invoking the operation *Fire* defined in the individual transition instances.

[3] When a state in the active state configuration is completed, a completion event is generated and enqueued into the *completionEventQueue* by invoking the operation *genCompletionEvent.*



$$genCompletionEvent \mathrel{\widehat=} [comEvent !: \downarrow Event] \fatsemi completionEventQueue.enqueue$$

$$[1]\ \langle \neg\ run\text{–}to\text{–}completion\rangle;\langle run\text{–}to\text{–}completion\rangle \subseteq \langle \neg\ dispatchEvent\rangle;(\langle dispatchEvent\rangle \cup \langle dispatchEvent\rangle;\langle true\rangle)$$

$$[2]\ \forall\ e{:}\downarrow Event \bullet \langle currentEvent = e\rangle;\langle currentEvent \neq e\rangle \subseteq \langle \neg\ fireTransitions\rangle;(\langle fireTransitions\rangle \cup \langle fireTransitions\rangle;\langle true\rangle)$$

$$[3]\ \forall\ s{:}\ activeStateConfiguration \bullet \langle \neg\ s.completed\ \rangle;\langle\ s.completed\rangle \subseteq \langle \neg\ genCompletionEvent\ \rangle; (\langle genCompletionEvent\rangle \cup \langle genCompletionEvent\rangle;\langle true\rangle)$$

5. Conclusion and Future Work

In this paper, we have presented a formal denotational and operational semantics for the UML state machine. The abstract syntax and the static and dynamic semantics of a model construct are formalized within a single Object-Z class. In particular, the dynamic semantics of the state machine are first defined in a denotational way by introducing semantic variables and invariants ignoring the operation sequences. The operational semantics of the state machine is defined in terms of Object-Z class operations and invariants defined using the timed refinement calculus. With this approach, we provide a complete formal semantics of the UML state machine that combines the denotational and operational semantics in a fully compositional manner by composing the semantics of its components.

Another significance of our work is that we deal with the UML state machine as part of the entire UML metamodel, so that the formal model given for the state machine should be interpreted in the context of other model elements in the metamodel. This provides a consistent semantic basis, which can be used for checking inconsistencies between different UML models and the completeness of the UML models.

The Object-Z expressions introduced in this paper are designed to be as descriptive as those are in an informal (or semi-formal) specification notation such as OCL. The result demonstrates that using a well-defined object-oriented formal specification technique like Object-Z for defining the UML metamodel neither increases the complexity nor decreases the readability. Instead, it enables the current UML metamodel to be viewed in an alternative way which is unambiguous, but still descriptive.

Acknowledgments

The authors thank Graeme Smith, Colin Fidge, Roger Duke and David Leadbetter for many helpful comments on a previous version of this paper.

References

[1] J. S. Dong and R. Duke, The Geometry of Object Containment, *Object-Oriented Systems*, vol. 2(1), pp. 41-63, Chapman & Hall, 1995.

[2] B. P. Douglass, *Real -Time UML: Developing Efficient Objects for embedded systems*, Addison-Wesley, 1998.

[3] R. Duke and G. Rose, *Formal Object-Oriented Specification Using Object-Z*, Macmillan, 2000.

[4] S. Dupuy, Y. Ledry, and M. Chabre-Peccoud, Translating the OMT Dynamic Model into Object-Z, *ZUM '98*, LNCS, No. 1493, pp. 347 - 366, 1998.

[5] R. France, A. Evans, K. Lano, and B. Rumpe, Developing the UML as a Formal Modeling Notation, *Computer Standards and Interfaces*, No. 19, pp. 325-334, 1998.

[6] R. France, A Problem-Oriented Analysis of Basic UML Static Modeling Concepts, *Proc of OOPSLA'99,* ACM SIGPLAN Notices, vol. 34, No. 10, October 1999.

[7] D. Harel, A. Pnueli, J.Schmidt, and R. Sherman, On the formal semantics of Statecharts, *Proc Symposium on Logic in Computer Science (LICS'87)*, pp. 56-64, 1987.

[8] D. Harel and A. Naamad, The STATEMATE semantics of Statecharts, *ACM Transaction on Software Engineering*, vol. 5, pp 293-333, 1996.

[9] B. Henderson-Sellers and F. Barbier, Black and White Diamonds, *Proc 2nd IEEE conference on UML: UML'99*, LNCS, No. 1723, pp. 550-565, 1999.

[10] C. Huizing, R. Gerth, and W.P. de Roever, Modeling Statecharts behavior in a fully abstract way, *in 13th Colloquium on Trees in Algebra and Programming*, LNCS, No. 299, pp. 271-294, 1988.

[11] S. Kent, A. Evans and B. Rumpe, UML Semantics FAQ, *ECOOP'99: Object-Oriented Technology*, LNCS, No. 1743, pp. 33-56, 1999.

[12] S-K. Kim and D. Carrington, Formalizing the UML class diagram using Object-Z, *Proc 2nd IEEE conference on UML: UML'99*, LNCS, No. 1723, pp. 83-98, 1999.

[13] S-K. Kim and D. Carrington, A Formal Mapping between UML Models and Object-Z Specifications, *ZB2000*, LNCS, No. 1878, pp. 2-21, 2000.

[14] S-K. Kim and D. Carrington, A Formal Denotational Semantics of UML in Object-Z, *the special issue of the journal of l'Objet*, To appear, 2000.

[15] S-K. Kim and D. Carrington, UML Metamodel Formalization with Object-Z: the State Machine Package, Technical Report 00-29, SVRC, The University of Queensland, Australia, 2000.

[16] D. Latella, I. Majzik, and M. Massink, Towards a formal operational semantics of UML statechart diagrams, *In 3rd International Conference on Formal Methods for Open Object-Oriented Distributed Systems(FMOODS)*, Kluwer, 1999.

[17] J. Lilius and I. P. Paltor, Formalizing UML state machines for model checking, Proc 2nd IEEE conference on UML: *UML'99*, LNCS, No. 1723, pp. 430-445, 1999.

[18] Object Management Group, OMG *Unified Modeling Language Specification*, version 1.3, 1999, http://www.omg.org

[19] A. Pnueli and M. Shalev, What is in a step: On the semantics of Statecharts, *in Theoretical Aspects of Computer Software (TACS'91)*, LNCS, No. 526, pp. 224-264, 1991.

[20] G. Smith and I. Hayes, Structuring Real-Time Object-Z Specifications, *IFM'2000*, LNCS, To appear, 2000.

[21] G. Smith. *The Object-Z Specification Language. Advances in Formal Methods*. Kluwer Academic Publishers, 2000.

[22] J. M. Spivey. *The Z Notation: A Reference Manual*, Prentice Hall, 2nd edition, 1992.

[23] A. Uselton and S. Smolka, A compositional semantics for Statecharts using labeled transition systems, *In CONCUR'94*, LNCS, No. 836, pp. 2-17, 1994.