SOFTWARE VERIFICATION RESEARCH CENTER
THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 02-23

A Formal Metamodeling Approach to a Transformation between Visual and Formal Modeling Techniques

Soon-Kyeong Kim and David Carrington

March 2002

Phone: +61 7 3365 1204
Fax: +61 7 3365 1999
http://svrc.it.uq.edu.au

A Formal Metamodeling Approach to a Transformation between Visual and Formal Modeling Techniques

Soon-Kyeong Kim and David Carrington

School of Information Technology and Electrical Engineering
The University of Queensland, Brisbane, 4072, Australia
Email: soon@itee.uq.edu.au, davec@itee.uq.edu.au

Abstract. Formal modeling notations and visual modeling notations can complement each other when developing software models. The most frequently adopted approach is to define transformations between the visual and formal models. However, a significant problem with the currently suggested approaches is that the transformation itself is often described imprecisely, with the result that the overall transformation task may be imprecise, incomplete and inconsistent. This paper presents a formal metamodeling approach to transform between UML and Object-Z. In the paper, the two languages are defined in terms of their formal metamodels, and a systematic transformation between the models is provided at the meta-level in terms of formal mapping functions. As a consequence, we can provide a precise, consistent and complete transformation between a visual model in UML and a formal model in Object-Z.

1. Introduction

Visual modeling techniques provide an opportunity to develop specifications that are easy to understand. Most visual modeling techniques, however, have imprecise modeling concepts and notations, and lack any systematic support for rigorous analysis of their models [8, 10, 15]. On the other hand, formal modeling techniques have advantages for producing a precise and analyzable specification, and to verify the properties of the specified system before implementation. Despite their potential, however, formal notations are considered difficult to use and understand [13, 19].

The most frequently adopted approach to overcome these deficiencies in both types of techniques is to define transformations between the visual and formal models [4, 6, 9, 16, 19, 20]. However, two significant problems with the currently suggested approaches are that the languages used often do not have a precise description for their syntax and semantics, and the transformation itself is often described imprecisely. Moreover, it is often difficult to verify the transformation is correct in terms of the transformation rules given. This means that the overall transformation task may be imprecise, incomplete and inconsistent. An incomplete and inconsistent transformation can cause unexpected behavioral consequences [9]. Consequently, the confidence

the developer may have in the models is reduced, making the transformation approach unreliable.

In order to enhance these problems, this paper introduces a formal metamodeling approach to integrate the two languages: UML [17] and Object-Z [3, 18]. In this work, we first formalize the UML metamodel using Object-Z. We also develop a formal metamodel of Object-Z adopting the same metamodeling architecture used for the UML metamodel. Given these metamodels, we then define a systematic transformation between these two languages at the meta-level in terms of formal transformation rules (Fig. 1). In this way, we not only give a precise description of the two languages but also provide a rigorous way to analyze UML models via a systematic transformation between the languages. Any verification of UML models can take place on their corresponding Object-Z specifications using reasoning techniques provided for Object-Z.

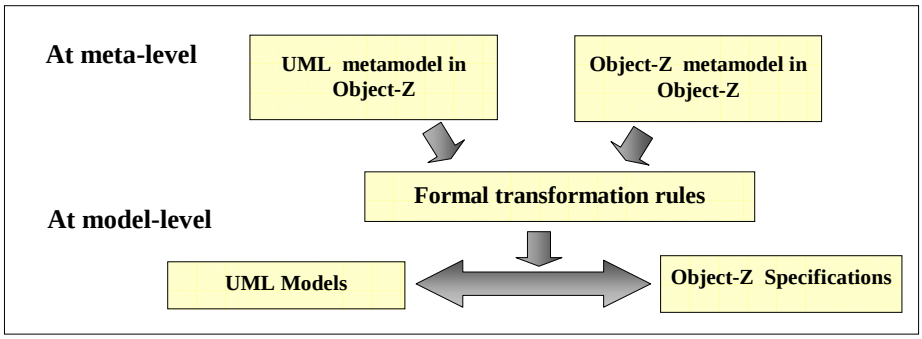


Figure 1. The approach taken in the paper

When we map a language to another, it is quite common to extend the target language in order to preserve the structural information of the source language during the mapping. For this reason, we extend the Object-Z metamodel presented in [11] to map UML modeling concepts as needed. This extension, however, should also contribute to enhancing Object-Z resulting in a more mature object-oriented modeling language. The advantages of the metamodel-based transformation can be summarized as follows: the transformation is defined in a systematic way at the meta-level, not the model-level; the semantic and syntactic structure is preserved during the transformation; inconsistency and incompleteness of the transformation can be verified in a systematic manner based on the metamodels of the languages; since the syntactic structure is preserved during the transformation, a systematic trace between the models in the two different languages is possible; and when the metamodel of a language is incomplete in terms of its semantics, mapping the language to another provides an extended semantic domain of that language.

Another significance of this work is that in our approach the entire transformation process is formalized. For example, the metamodels of both languages are formally defined using Object-Z. Given the formal metamodels, the transformation rules are also formally defined. This feature not only makes our approach precise but also increases the confidence the developer may have in the models developed using the transformation approach.

Due to space limitations, in this paper we focus on a mapping between the UML state machine and Object-Z and refer readers to [10, 11] for a mapping between the UML class diagram and Object-Z classes. It should be also noted that in this paper it is not our intention to show how the transformed Object-Z model of a UML model can be used for rigorous analysis of the UML model, rather we focus on describing our metamodel-based (formal) transformation approach precisely.

The structure of the rest of this paper is as follows. In Section 2 we present a formal model of the UML metamodel in Object-Z. In Section 3 we present a metamodel of Object-Z extending our previous work [13]. In Section 4 given the formal metamodels of both languages, we introduce a formal mapping between the two languages. Finally, Section 5 draws some conclusions.

2. A formal model of the UML metamodel

The UML metamodel [17] is developed from three abstract views: the abstract syntax, static semantics and dynamic semantics. Each view is represented separately in a different representation with consequently many redundancies and inconsistencies [5, 7, 12]. In our work, these three views of each distinct modeling construct are gathered together and captured by a single Object-Z class. Object-Z has better expressiveness than class diagrams, so it can encapsulate context conditions (well-formedness rules) with the syntax, while keeping the same structure as the class diagrams in terms of classes. Fig. 2 is a graphical description for this. With this approach, our approach not only provides a formal specification of the UML metamodel but also overcomes the lack of modularity and extensibility of the current UML semantic representation (see [12, 14] for the advantages of using Object-Z formalizing UML).

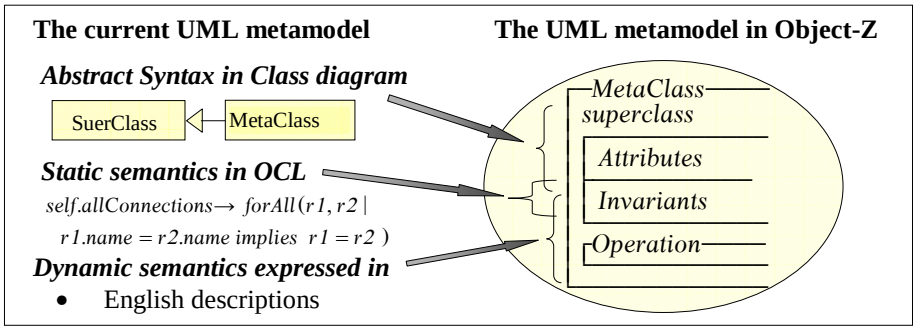


Figure 2. A graphical description showing how to group the three different views

2.1 A formal model of the UML state machine

The State Machine package in UML consists of the modeling concepts (Fig. 3) used to specify the behavior of various dynamic model elements, which are viewed as state-chart diagrams [2, 17]. In this paper, we present only a (part) formal description of the core model elements of the UML state machine (see [14] for a full formal description of the UML metamodel).

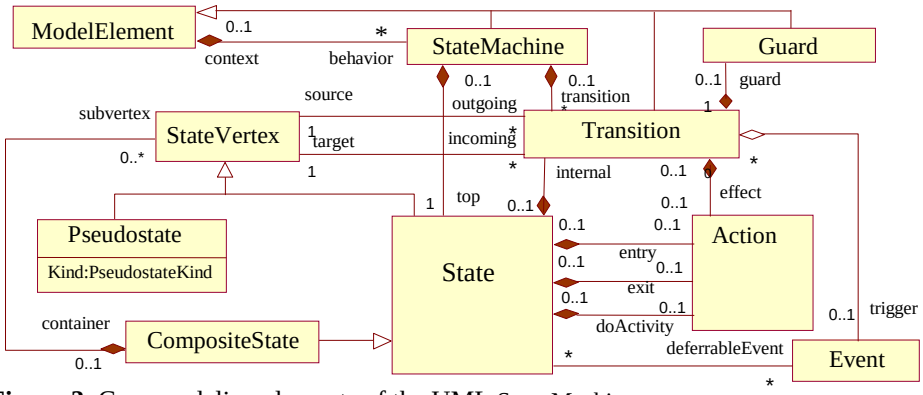


Figure 3. Core modeling elements of the UML State Machine

State: Syntactically State inherits from StateVertex and has several associations with StateMachine, Action, Event and Transition: *stateMachine*, *entry*, *doActivity*, *exit*, *deferrableEvent*, and *internal* (Fig. 3). The following Object-Z class is a formal description of State. StateVertex is included as a superclass (we assume that StateVertex is also formalized as an Object-Z class with the same name) and the associations are formalized as attributes.

State
StateVertex

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

Event and Action: Event and Action inherit from ModelElement. Event has parameters and associations to Transition and State. Action has an attribute *isAsynchronous* indicating whether or not the action is asynchronous and several other attributes (refer to [14] for details).

Event
ModelElement

$$
\begin{array}{l}
transition : \mathbb{P}\ Transition \\
state : \mathbb{P}\downarrow State \\
parameters : seq\ Parameter\ \copyright\ \ldots
\end{array}
$$

Action
ModelElement

$$
\begin{array}{l}
isAsynchronous : Boolean \\
entryState : \mathbb{P}\ State \\
exitState : \mathbb{P}\ State \\
activityState : \mathbb{P}\ State\ \ldots
\end{array}
$$

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

PseudostateKind:=*initial* | *deepHistory* | *shallowHistory* | *join* | *fork* | *junction* | *choice*

Pseudostate
StateVertex

kind : *PseudostateKind*
…

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 [17]. It inherits from State and has a composition association to StateVertex. The static semantics (e.g. there must be at least two composite substates in a concurrent composite state) are formalized as invariants in the predicate of the following Object-Z class.

CompositeState
State

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

isConcurrent $\Rightarrow$ #{ *s*: *subvertex* | *s* $\in$ *CompositeState* } $\geq$ *2*
…

Transition: Transition has a source and a target state vertex. It also can have a guard, a trigger event, effect actions, and a state for internal transitions.

Transition
ModelElement

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

…

StateMachine: StateMachine has composition relationships to State and Transition, and an aggregation relationship to ModelElement which is the context of the state machine.

StateMachine
ModelElement

context : $\mathbb{P}\downarrow$ *ModelElement*
top : $\downarrow$ *State* ©
transition : $\mathbb{P}$ *Transition* ©

…

3. The Object-Z metamodel

Fig. 4 is a UML class diagram showing the abstract syntax of core modeling constructs in Object-Z. For brevity, we focus on attributes and operations in this paper (refer to [11, 13] for details). We also assume that all types used in this paper are already defined as distinct Object-Z classes.

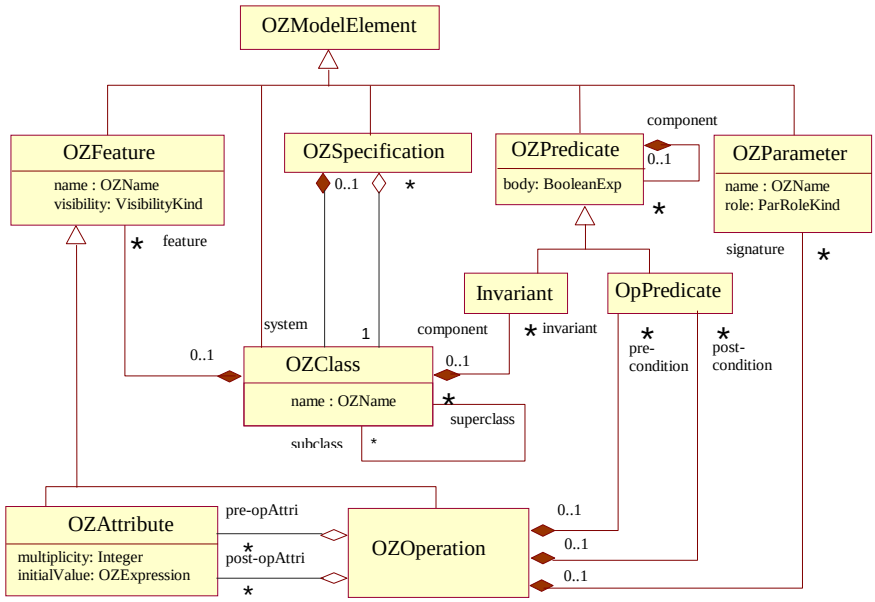


Figure 4. A class diagram showing the structure of core model elements in Object-Z

The following Object-Z class *OZClass* is a formal description for classes in Object-Z. The attribute *superclass* maintains inheritance information. Each class has its own features, i.e. attributes and operations defining static and dynamic behaviors of its instances.

OZClass

OZModelElement

$$superclass : \mathbb{P}\ OZClass$$
$$features : \mathbb{P}\downarrow OZFeature\ ©$$
$$invariant : \mathbb{P}\ Invariant\ ©$$

Attributes: Object-Z attributes can be further classified into pure and relationship attributes depending on their roles (Fig. 5). Pure attributes are those not modeling relationships between classes. On the other hand, relationship attributes model relationships between classes using the instantiation mechanism in Object-Z. Like UML, relationships between objects can be common reference relationships, shared or unshared whole-part relationships. For this, we define an enumeration type, *RelationshipKind*, which has *reference*, *sharedOwner* and *unsharedOwner* as its values.

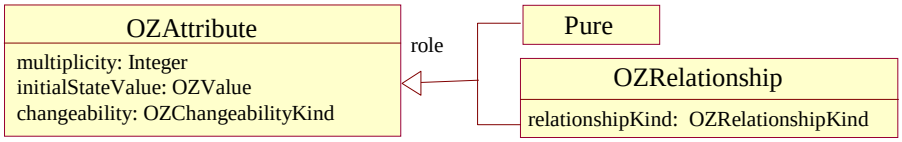
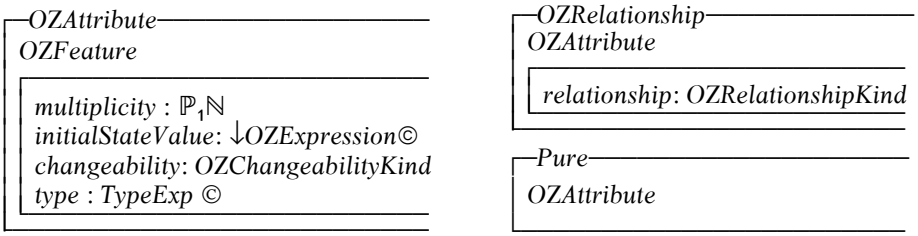


Figure 5. A class diagram showing a classification of attributes in Object-Z



Operations: Operations in Object-Z can be classified as local or interaction (Fig. 6). Local operations model the local behavior of objects. Interaction operations model interactions with other objects. Obviously, interaction operations are related to relationship attributes.

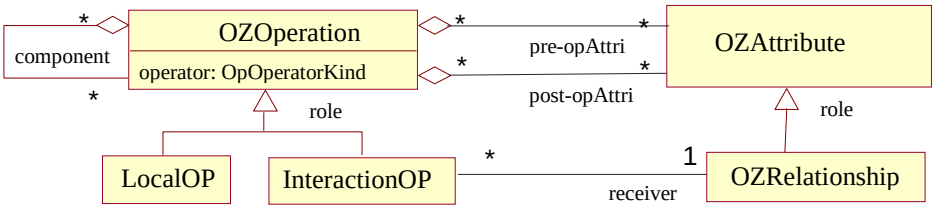
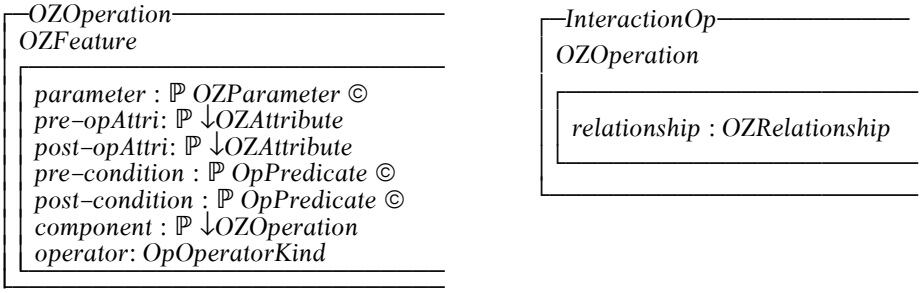


Figure 6. A class diagram showing a classification of operations in Object-Z



4. A Formal mapping between UML and Object-Z

In [11], we describe a formal mapping between the static view of a system modeled by UML class diagrams and Object-Z classes. In this paper, we present a formal mapping between the dynamic view of a system modeled by UML state machines and Object-Z.

The transformation rules presented in this section are based on the formal description of the UML state machine in Section 2 and the Object-Z metamodel presented in Section 3.

4.1. Transformation rules for state machines

When the context of a state machine is a class, the state machine as a whole describes the behavior of the class (Fig. 7). In Object-Z the behavior of a class (or, to be precise, an object of the class) can be modeled in terms of its class attributes and operations. For example, in Object-Z an object state is modeled by class attributes which together denote a distinct (behavioral) state of the object. Then object behavior can be defined in the context of class attributes and the class operations (events) that change the values of these attributes (transitions). This example clearly demonstrates the semantic comparability between the UML state machine and the object behavior defined in Object-Z in terms of class attributes and operations (Fig. 7 shows this semantic mapping). To provide a direct syntactical mapping between the two languages, however, the syntactic structure of the Object-Z metamodel presented in Fig. 5 is extended according to that of the UML state machine. This enables us to preserve the syntactic structure of the two languages during the transformation and makes the translation process systematic and precise. Detailed transformation rules are as follows.

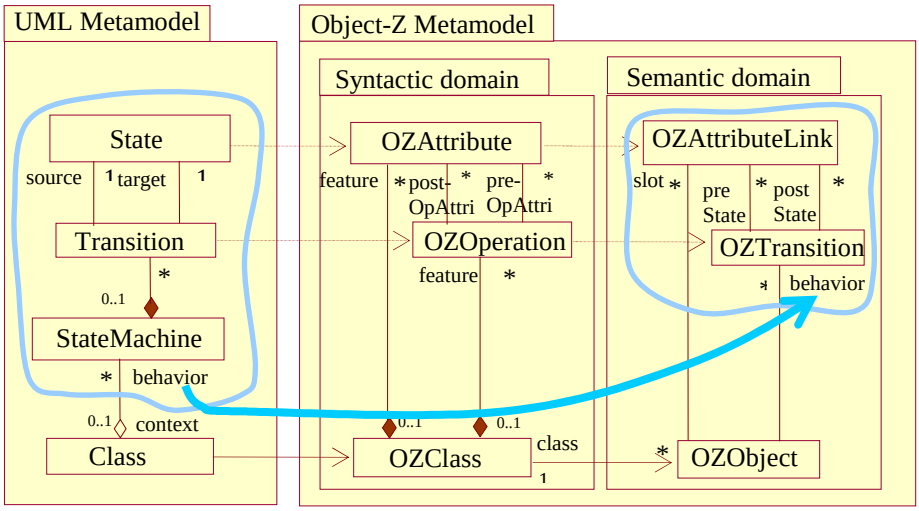


Figure 7. Semantic comparability between UML State machines and Object-Z

States : A state in the state machine is a condition during the lifetime of an object. The condition can be either a passive situation, e.g. an object waiting for some event to occur, or an active situation, e.g. the object is performing some actions or activities. As claimed, in Object-Z such a behavioral state of an object can be modeled with a class attribute. Although standard Object-Z does not explicitly distinguish whether or not an attribute models a static or a behavioral state of the object, in order to provide a direct syntactical mapping between the two languages, we extend the attribute struc-

ture of Object-Z by further classifying pure attributes into static or behavioral state attributes. Behavioral state attributes model the same notion of states in the state machine, e.g. capturing a situation in which the object is doing or waiting for some actions (see Fig. 8 for this extension).

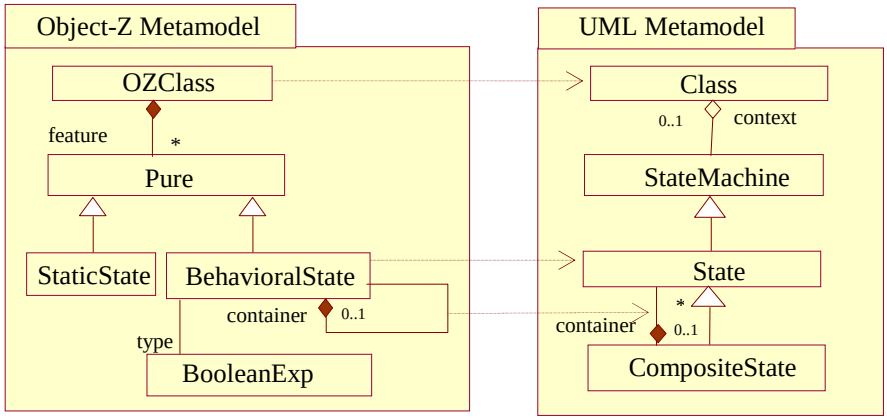


Figure 8. A class diagram showing an extended structure of attributes in Object-Z

Since behavioral attributes model observable states of objects, they are visible. The possible values of the attributes are Boolean values. When a behavioral state attribute is *true*, it means that the object is in that behavioral state, which is regarded as an *active* state in UML. Since states in the state machine can be contained by a composite state, the metaclass BehavioralState has an attribute *container* of type itself. We also define an attribute *isConcurrency* of type StateConcurrencyKind to formalize the concept of composite states in the state machine and their concurrency. These attributes are used to formalize the static semantics of the UML state. A formal description of the metaclass BehavioralState is given below.

$StateConcurrencyKind ::== simple \mid concurrentComp \mid nonConcurrentComp$

BehavioralState
Pure

$container : \mathbb{P}\, BehavioralState \,©$
$isConcurrency : StateConcurrencyKind$

$\#container \leq 1 \wedge type \in BooleanType \wedge visibility = visible$
$type.evaluate = true \Rightarrow$
$[1]\ (\forall c : container \bullet c.evaluate = true)$
$[2]\ isConcurrent = concurrentComp \Rightarrow$
$(\forall s : BehavioralState \mid self \in s.container \bullet s.type.evaluate = true)$
$[3]\ isConcurrent = nonConcurrentComp \Rightarrow$
$\#\{s : BehavioralState \mid self \in s.container \wedge s.type.evaluate = true\} = 1$

The invariants defined in the predicate formalize the static semantics of states:
[1] When a state is *true* meaning active in the UML terminology, its container state should be also *true*.

[2] For a concurrent composite state, when it is *true*, all its containing states should be *true*.

[3] For a non-concurrent composite state, when it is *true*, only one of its sub states should be *true* at one time.

The state hierarchy is also formalized by these invariants.

We now map each state of a state machine to a behavioral state attribute in Object-Z. Since in UML, states contained in different composite states can have the same name, we define an auxiliary function *convStateName* that returns a unique name for each behavioral state attribute defined in an Object-Z class. In this way, we ensure that attribute names defined in an Object-Z class are unique. The function *mapUMLStateToOZ* takes a UML state and returns a behavioral attribute of Object-Z. The corresponding behavioral state attribute of the UML container becomes the container of the Object-Z behavioral state attribute.

$$convStateName : Name \rightarrow OZName$$

$$mapUMLStateToOZ : \downarrow State \rightarrow BehavioralState$$

$$\forall\, us: \downarrow State, os: BehavioralState \bullet$$
$$mapUMLStateToOZ(us) = os \iff os.name = convStateName(us.name) \land$$
$$os.container = \{uc: us.container \bullet mapUMLStateToOZ(uc)\}$$

We now map all states defined in a state machine to Object-Z. We restrict the context of the state machine to a UML class defined within a given UML class diagram. In this context, each state defined in the state machine maps to a distinct behavioral attribute of the Object-Z class corresponding to the UML class. The function *mapUMLStateMachineToOZ* is a formal description of this rule and it is defined compositionally using the pre-defined function for states. The constraint defined in the function ensures the completeness of the Object-Z class with respect to states of the state machine. It should be noted that the function *mapUMLStateMachineToOZ* is not completely defined yet and it is extended in later sections of this paper with respect to other model elements of the state machine, e.g. transitions, events, and actions (also see [11] for the structure of an Object-Z class *UMLClassDiagram*).

$$mapUMLStateMachineToOZ : UMLStateMachine \rightarrow OZClass$$

$$\forall\, sm: StateMachine, d: UMLClassDiagram \mid sm.context \subseteq d.classes \bullet$$
$$(\forall\, s: OZSpecification \mid mapUMLClassDiagramToOZSpec(d) = s \bullet$$
$$(\exists\, oc: s.classes \mid \{oc\} = \{uc: sm.context \bullet mapUMLClassToOZ(uc)\} \bullet$$
$$mapUMLStateMachineToOZ(sm) = oc \iff$$
$$(\forall\, st: sm.stateConfiguration \mid st \in \downarrow State \bullet$$
$$(\exists_1\, ba : oc.feature \mid ba \in BehavioralState \bullet$$
$$MapUMLStateToOZ(st) = ba)) \land$$
[Completeness of the Object-Z class]
$$\{st: sm.stateConfiguration \mid st \in \downarrow State \bullet MapUMLStateToOZ(st)\} =$$
$$\{ba : oc.feature \mid ba \in BehavioralState\})) \ \ldots$$

Events : An event represents the reception of a signal or a request to invoke an operation (a call event) [17]. From an object's point of view, responding to such a request should be modeled as an operation (we call this operation an *event acceptor operation*). Consequently, we transform each event into an event acceptor operation. Since

the reception of an event is a local behavior of the receiving object, event acceptor operations inherit from local operations in the Object-Z metamodel (see the metaclass EventAccptOP in Fig. 9).

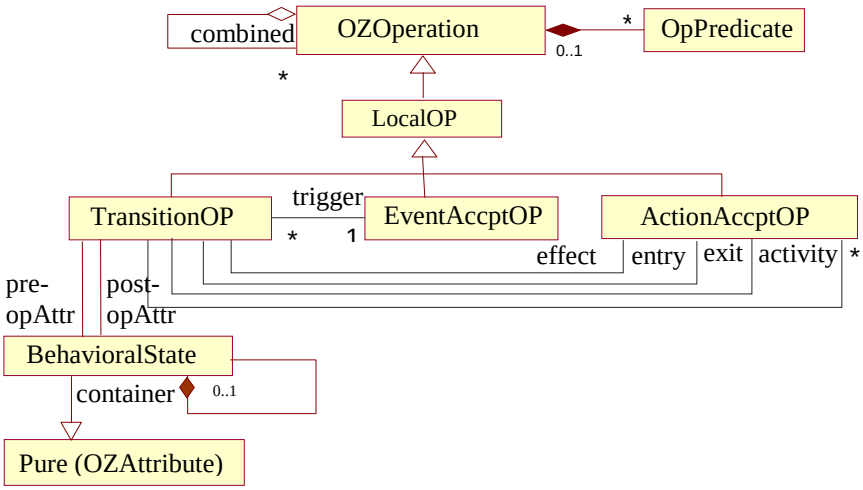


Figure 9. A class diagram showing an extended structure of operations in Object-Z

Prior to formalizing event acceptor operations, we extend the Object-Z class *OZOperation* defined in Section 3 as follows. The secondary attribute called *allComponent* holds all operation components recursively used to define the operation.

OZOperation

…
Δ
$allComponent : \mathbb{P}\ OZOperation$

$allComponent = component \cup \bigcup \{c: component \bullet c.allComponent\}$

The following Object-Z class *EventAccptOP* is a formal description of the metaclass EventAccptOP. Since the reception of an event results in firing transitions, an event acceptor operation contains a set of operations defined for the transitions.

EventAccptOP
LocalOP

$triggeredTranOp : \mathbb{P}\ TransitionOP$

$triggeredTranOp = \{c: allComponent \mid c.component = \varnothing\}$

We now map each event of a state machine to an event acceptor operation of Object-Z. The function *mapUMLEventToOZ* takes a UML event and returns an event acceptor operation of Object-Z. The parameters of the event map to those of the event acceptor operation. The Object-Z operations corresponding to the transitions that the event fires are defined as the triggered transition operations of the event acceptor operation (see the Object-Z class *TransitionOp* for the definition of transition operations and the

mapping function *mapUMLTransitionToOZ* in later this section). The conditions of change events are formalized as the pre-condition of their event acceptor operations. When an event triggers more than one transition, their transition operations are combined by the following rules within the event acceptor operation.

When the source states of transitions are contained in the same composite state (note that in this case the composite state is always non-concurrent), only one of the transitions can be fired at any time. Consequently, we combine the transition operations using the choice operator ([]) in Object-Z. Therefore, there should exist a component operation in the set *allComponent* of the event acceptor operation that combines the transition operations using the choice operator ([]).

When the source states of transitions are contained in different composite states, the transition operations are combined depending on the concurrency of the least common ancestor of the source states. For example, when the least common ancestor of two transitions is concurrent, the transition operations are combined using the conjunction operator ($\wedge$). Otherwise, the operations are combined using the choice operator. Since Object-Z operations are combined recursively using other operations, the operations actually combined as a component in the set *allComponent* of the event acceptor operation are those containing not only the transition operations of the two transitions but also the transition operations of all other transitions defined in the state configuration of the least common ancestor and also fired by the event. In this way, we can formalize transitions triggered by the same event with respect to their state hierarchy and concurrency.

We assume that the function *leastComAncestor* (which returns the least common ancestor of the source states of two transitions) and the function *convExpression* (which converts UML expressions to Object-Z expressions) are defined.

$$
\begin{array}{|l}
mapUMLEventToOZ : \downarrow Event \rightarrow EventAccptOP \\
\hline
\forall\, ue{:}\downarrow Event, ep{:}\, EventAccptOP \bullet mapUMLEventToOZ(ue) = ep \Longleftrightarrow \\
\quad ep.name = convName(ue.name) \wedge \\
\quad \{p{:}\, ue.parameter \bullet convParameter(p)\} = ep.parameter \wedge \\
\quad \{t{:}\, ue.transition \bullet mapUMLTransitionToOZ(t)\} = ep.triggeredTranOp \\
\quad \textbf{[\textit{Transitions originating from the same composite state}]} \\
\quad (\forall\, t{:}\, ue.transition \bullet \\
\qquad \{ot : ue.transition \setminus \{t\} \mid ot.source.container = t.source.container\} \neq \varnothing \Longrightarrow \\
\qquad (\exists_1\, eo{:}\, ep.allComponent \bullet \\
\qquad\quad eo.allComponent = \{mapUMLTransitionToOZ(t)\} \cup \{ot : ue.transition \setminus \{t\} \\
\qquad\quad ot.source.container = t.source.container \bullet mapUMLTransitionToOZ(ot)\} \wedge \\
\qquad\quad \{Choice\} = \{e{:}\, eo.allComponent \bullet e.operator\})) \\
\quad \textbf{[\textit{Transitions originating from different composite states}]} \\
\quad (\forall\, t1, t2{:}\, ue.transition \mid t1 \neq t2 \wedge t1.source.container \neq t2.source.container \wedge \\
\qquad t1.source.allContainingStates \cap t2.source.allContainingStates \neq \varnothing \bullet \\
\qquad (\exists_1\, eo{:}\, ep.allComponent \bullet \\
\qquad\quad (\forall\, eeo1, eeo2 : eo.component \mid eeo1 \neq eeo2 \bullet \\
\qquad\quad eeo1.allComponent = \{mapUMLTransitionToOZ(t1)\} \cup \\
\qquad\qquad \{ot{:}\, ue.transition \mid t1 \neq ot \wedge \\
\qquad\qquad\quad ot.source \in \cup\{s{:}\, t1.source.allContainingStates \cap \\
\qquad\qquad\quad leastComAncestor(t1, t2).stateCongifuration \bullet s.stateConfiguration\} \bullet \\
\qquad\qquad\quad mapUMLTransitionToOZ(ot)\} \\
\qquad\quad eeo2.allComponent = \{mapUMLTransitionToOZ(t2)\} \cup
\end{array}
$$

$$
\begin{array}{l}
\quad \{ot: ue.transition \mid t2 \neq ot \wedge \\
\quad ot.source \in \bigcup \{s: t2.source.allContainingStates \cap \\
\qquad leastComAncestor(t1, t2).stateConfiguration \bullet s.stateConfiguration\ \} \bullet \\
\qquad\quad mapUMLTransitionToOZ(ot)\}) \\
\quad leastComAncestor(t1, t2).isConcurrent \Rightarrow eo.operator = Conjunction \\
\quad \neg leastComAncestor(t1, t2).isConcurrent \Rightarrow eo.operator = Choice\ )) \\
ue \in ChangeEvent\ \Rightarrow convExpression(ue.changeExpression) \in \\
\quad \{p: ep.pre\text{-}condition \bullet p.body\}
\end{array}
$$

Actions: An action is a specification of an executable statement in UML [17], so we formalize actions in UML as operations in Object-Z (we call these operations *action acceptor operations*). We extend the Object-Z metamodel accordingly (see Fig. 9).

ActionAccptOP

$$LocalOP$$

The function *mapUMLActionToOZ* takes a UML action and returns an action acceptor operation of Object-Z. When a call action invokes an operation defined in the same class, for brevity we directly map the Object-Z operation corresponding to the operation as the action acceptor operation. Note that it is assumed that the Object-Z class *Action* defined in section 2 now has an additional attribute called *target* of type Instance (see [12]) which holds the target instances of the action. Also note that since the argument and script of actions are expressions which can be defined using any description language, no further rules are given for these constructs.

$$
\begin{array}{l}
mapUMLActionToOZ : \downarrow Action \rightarrow \downarrow OZOperation \\
\hline
\forall\ ua: \downarrow Action, ap: \downarrow OZOperation \bullet mapUMLActionToOZ(ue) = ap \Leftrightarrow \\
\quad ua \in CallAction \wedge ua.target \subseteq \bigcup \{o: ua.operation.owner \bullet o.instance\} \Rightarrow \\
\qquad ap = mapUMLOperationToOZ(ua.operation) \\
\quad \neg(ua \in CallAction \wedge ua.target \subseteq \bigcup \{o: ua.operation.owner \bullet o.instance\})\ \Rightarrow \\
\qquad (\exists\ op : ActionAccptOP \mid op.name = convName(us.name) \bullet ap = op)
\end{array}
$$

Transitions: A transition has two aspects. First, it presents a change in the state of an object. Second, it also presents the execution of actions, e.g., exit, entry, or effect actions, or activities associated with the transition. We formalize the object behavior presented by transitions using class operations in Object-Z (we call these operations *transition operations*). Fig. 9 shows this extension to the Object-Z metamodel. The following Object-Z class is a formal description of transition operations.

TransitionOP

$$
\begin{array}{l}
LocalOP \\
\hline
source : BehavioralState \\
target : \mathbb{P}\ BehavioralState \\
stateExit : seq \downarrow OZOperation \\
effect : seq \downarrow OZOperation \\
stateEntry : seq \downarrow OZOperation \\
stateActivity : seq \downarrow OZOperation \\
actionSequence : seq \downarrow OZOperation \\
\hline
source \in pre\text{-}opAttr \wedge target \subseteq post\text{-}opAttr \wedge \#\ effect = 1 \\
[1]\ stateExit \frown effect \frown stateEntry \frown stateActivity = actionSequence \\
[2]\ ran\ actionSequence = allComponent
\end{array}
$$

The Object-Z class *TransitionOP* has two behavioral state attributes *source* and *target* each of which represents the source and target states of the transition respectively. Since the source state is a condition to fire the transition, the attribute *source* is defined as a pre-operation attribute of the transition operation and its value is used as a pre-condition of the operation. Similarly, the attribute *target* is defined as a post-operation attribute and it value is used as a post-condition of the operation. The class has an action operation representing its effect and a set of action operations presenting the exit actions of the states in the full hierarchy of the source state (see the secondary attribute *explicitSourceState* in [14] for the concept of the state hierarchy). It also has two sets of action operations *stateEntry* and *stateActivity* presenting the entry actions and the activities of the states in the full hierarchy of the target state respectively. Finally, it includes an attribute called *actionSequence* formalizing the execution sequence of actions. The action operations are combined in the following sequence: the exit actions of the source states, the effect action stated in the transition, the entry actions stated in the target states, and the activities stated in the target states (see invariant [1]). The components of a transition operation are the action acceptor operations corresponding to the actions associated with the transition (see invariant [2]). Detailed transformation rules follow.

***State changes*:** The source and target states of a transition are used to define those of the transition operation. That is, the Object-Z behavioral state attribute corresponding to the source state of the transition is defined as the source state of the transition operation. On the other hand, the target state is transformed by the following rules:

When the target state of the transition is a simple state (not a composite state), its corresponding Object-Z behavioral state attribute is defined as the target state of the transition operation.

When the target state of the transition is a composite state, the initial state in the composite state or each of the concurrent regions (if the composite state is concurrent) is the target state of the transition unless the initial state is a history state. When the initial state or each of the concurrent regions is also a composite state, this rule applies to the rest of the full hierarchy of the target state (see the secondary attribute *explicitTargetState* in [14] for the concept of this state hierarchy). In this case, the final target state is the inner-most state (or states) in the full hierarchy of the initial state (or each of the concurrent regions) so that their corresponding Object-Z behavioral state attribute (or attributes) are defined as the target states of the transition operation. The rest of the state hierarchy of the target state (or states) is formalized with the behavioral state attributes. When entering a shallow or a deep history state (see [14] for the concept of history states), the inner-most states in the full hierarchy of the shallow or the deep state are used to define the target state of the transition operation (see the constraints defined in the function *mapUMLTransitionToOZ*).

Guards: When a guard condition exists, it is translated as the pre-condition of the transition operation.

Entry and exit actions and activities: For the exit actions of the source states and the entry and activity actions of the target states, their corresponding Object-Z action acceptor operations are defined as the exit, entry, and activity operations of the transition operation respectively.

Effect Action: If a transition has an effect action, its corresponding Object-Z action acceptor operation is defined as the effect operation of the transition operation.

We are now in a position to formalize all the rules defined above. We first extend the Object-Z class *CompositeState* defined in section 2 by defining three additional secondary attributes *allDefaultStates*, *allShallowHistoryStates*, and *allDeepHistoryStates* which return states in the full hierarchy of the initial state, the shallow or the deep history state respectively. When a transition enters a composite state, these attributes are used to define the target scope of the transition.

CompositeState

$\ldots$

$[Attributes]$

Δ

$allDefaultStates : \mathbb{P}\downarrow State$

$allShallowHistoryStates : \mathbb{P}\downarrow State$

$allDeepHistoryStates : \mathbb{P}\downarrow State$

We also define an auxiliary function *convTransName* which returns a unique name for each transition operation.

$convTransName : Transition \rightarrow OZName$

Finally, we formalize the transformation rules defined for transitions in terms of a formal function *mapUMLTransitionToOZ* as follows.

$mapUMLTransitionToOZ : Transition \rightarrow TransitionOP$

$\forall\ ut: Transition, tp: TransitionOP \bullet\ mapUMLTransitionToOZ(ut) = tp \Leftrightarrow$
$ut.name = convTransName(tp.name) \wedge$
[*Constraints for the source states*]
$mapUMLStateToOZ(ut.mainSource) = tp.source \wedge$
[*Constraints for exit and effect actions*]
$\{e: \cup\{s: ut.explicitSourceStates \bullet s.exit\} \bullet mapUMLActionToOZ(e)\} = ran\ tp.exit \wedge$
$\{a: ut.effect \bullet mapUMLActionToOZ(a)\} = ran\ tp.effect \wedge$
[*Guards as pre-conditions*]
$\{e: \cup\{g: ut.guard \bullet g.expression\} \bullet convExpression(e)\} \subseteq$
$\cup\{p: tp.pre\text{-}condition \bullet p.body\}$
[*Constraints for the target state when the main target is a simple state*]
$ut.mainTarget \in SimpleState \Rightarrow \{mapUMLStateToOZ(ut.mainTarget)\} = tp.target \wedge$
$\{e: \cup\{s: ut.explicitTargetStates \bullet s.entry\} \bullet mapUMLActionToOZ(e)\} =$
$ran\ tp.stateEntry \wedge$
$\{e: \cup\{s: ut.explicitTargetStates \bullet s.doActivity\} \bullet mapUMLActionToOZ(e)\} =$
$ran\ tp.stateActivity$
[*Constraints for the target state when the main target is a composite state*]
$ut.mainTarget \in CompositeState \Rightarrow$
[*Entering an initial state*]
$(\exists\ s: ut.mainTarget.initialVertex \mid s.kind = initial \bullet$
$\{s: ut.mainTarget.allDefaultStates \mid s \in Simple \bullet$
$mapUMLStateToOZ(s)\} = tp.target \wedge$
$\{e: \cup\{s: ut.mainTarget.allDefaultStates \mid s \in Simple \bullet s.entry\} \bullet$
$mapUMLActionToOZ(e)\} = ran\ tp.stateEntry \wedge$
$\{e: \cup\{s: ut.mainTarget.allDefaultStates \mid s \in Simple \bullet s.doActivity\} \bullet$
$mapUMLActionToOZ(e)\} = ran\ tp.stateActivity\ ) \vee$

[*Entering a shallow history state*]
$(\exists\, s: ut.mainTarget.initialVertex \mid s.kind = shallowHistory \bullet$
$\{s: ut.mainTatget.allShallowHistoryStates \mid s \in Simple \bullet$
$mapUMLStateToOZ(s)\} = tp.target \wedge$
$\{e: \cup \{s{:}ut.mainTarget.allShallowHistoryStates \mid s \in Simple \bullet s.entry\} \bullet$
$mapUMLActionToOZ(e)\} = ran\ tp.stateEntry \wedge$
$\{e: \cup \{s{:}ut.mainTarget.allShallowHistoryStates \mid s \in Simple \bullet s.doActivity\} \bullet$
$mapUMLActionToOZ(e)\} = ran\ tp.stateActivity\ ) \vee$
[*Entering a deep history state*]
$(\exists\, s: ut.mainTarget.initialVertex \mid s.kind = deepHistory] \bullet$
$\{s: ut.mainTarget.allDeepHistoryStates \mid s \in Simple \bullet$
$mapUMLStateToOZ(s)\} = tp.target \wedge$
$\{e: \cup \{s{:}ut.mainTarget.allDeepHistoryStates \mid s \in Simple \bullet s.entry\} \bullet$
$mapUMLActionToOZ(e)\} = ran\ tp.stateEntry \wedge$
$\{e: \cup \{s{:}ut.mainTarget.allDeepHistoryStates \mid s \in Simple \bullet s.doActivity\} \bullet$
$mapUMLActionToOZ(e)\} = ran\ tp.stateActivity\ )$

We now extend the function *mapUMLStateMachineToOZ* previously defined to transform actions, events and transitions as follows. Constraints ensuring the completeness of the Object-Z are added at the end of the function.

$mapUMLStateMachineToOZ : UMLStateMachine \rightarrow OZClass$

$\forall\, sm: StateMachine, d: UMLClassDiagram \mid sm.context \subseteq d.classes \bullet$
$(\forall\, s: OZSpecification \mid mapUMLClassDiagramToOZSpec(d) = s \bullet$
$(\exists\, oc: s.classes \mid \{oc\} = \cup\{uc: sm.context \bullet mapUMLClassToOZ(uc)\} \bullet$
$mapUMLStateMachineToOZ(sm) = oc \iff$
[*States*]
$(\forall\, st: sm.stateConfiguration \mid st \in \downarrow State \bullet \ldots$
[*Actions of the states*]
$(\exists_1\, ep : oc.feature \mid ep \in \downarrow OZOperation \bullet$
$\{ep\} = \{e: st.entry \bullet MapUMLActionToOZ(e)\})$
$(\exists_1\, xp : oc.feature \mid xp \in \downarrow OZOperation \bullet$
$\{xp\} = \{e: st.exit \bullet MapUMLActionToOZ(e)\})$
$(\exists_1\, ap : oc.feature \mid ap \in \downarrow OZOperation \bullet$
$\{ap\} = \{e: st.doActivity \bullet MapUMLActionToOZ(e)\}))$
[*Transitions*]
$(\forall\, t: sm.transition \bullet (\exists_1\, tp : oc.\ feature \mid tp \in TransitionOP \bullet$
$tp = MapUMLTransitionToOZ(t) \wedge$
$tp.source \in \{ba : oc.feature \mid ba \in BehavioralState\} \wedge$
$tp.target \subseteq \{ba : oc.feature \mid ba \in BehavioralState\} \wedge$
[*Actions, effects and events on the transitions*]
$ran\ tp.stateEntry \subseteq \{ap : oc.feature \mid ap \in \downarrow OZOperation\} \wedge$
$ran\ tp.stateExit \subseteq \{ap : oc.feature \mid ap \in \downarrow OZOperation\} \wedge$
$ran\ tp.stateActivity \subseteq \{ap : oc.feature \mid ap \in \downarrow OZOperation\}) \wedge$
$(\forall\, e: t.effect \bullet MapUMLActionToOZ(e) \in ran\ tp.effect))$
$(\forall\, e: t.trigger \bullet (\exists_1\, ep : oc.\ feature \mid ep \in EventAccptOP \bullet$
$ep = MapUMLEventToOZ(e) \wedge tp \in ep.triggeredTransOP)))$
[*Completeness of the Object–Z class*]
…
$\{t: sm.transition \bullet MapUMLTransitionToOZ(t)\} =$
$\{tp : oc.feature \mid tp \in TransitionOP\}$
$\{e: \cup \{t: sm.transition \bullet t.trigger\} \bullet MapUMLEventToOZ(e)\} =$
$\{ep : oc.feature \mid ep \in EventAccptOP\}$
$\{ap : oc.feature \mid ap \in ActionAccptOP\} \subseteq$

$$
\begin{aligned}
&\{a: \cup \{s{:}sm.stateConfiguration \mid s \in \downarrow State \bullet s.entry\} \bullet \\
&\quad MapUMLActionToOZ(a)\} \cup \\
&\{a: \cup \{s{:}sm.stateConfiguration \mid s \in \downarrow State \bullet s.exit\} \bullet \\
&\quad MapUMLActionToOZ(a)\} \cup \\
&\{a: \cup \{s{:}sm.stateConfiguration \mid s \in \downarrow State \bullet s.doActivity\} \bullet \\
&\quad MapUMLActionToOZ(a)\} \cup \\
&\{a: \cup \{t{:}\ sm.transition \bullet t.effect\} \bullet\ MapUMLActionToOZ(a)\}\ ))
\end{aligned}
$$

4.2 Example

The example presented in this section is a simplified light control system [1]. The system structure is modeled using the class diagram (see Fig. 10). Invariants and operation specifications in OCL [17] are added to the class LightGroup.

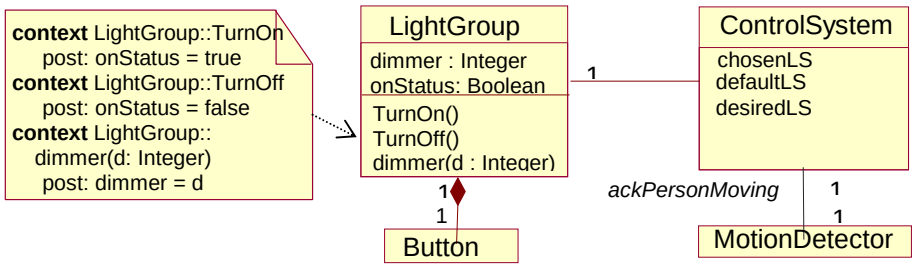
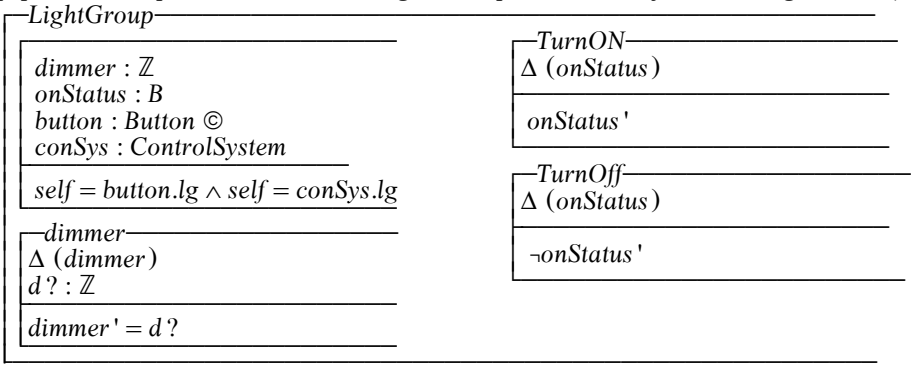


Figure 10. Class diagram for the light control system

Formalizing the static part of the system: We formalize the class diagram in Fig. 10 according to the transformation rules presented in [11]. Each UML class in the diagram maps to a distinct Object-Z class and associations between the UML classes are formalized as relationship attributes in the corresponding Object-Z classes. In particular, the operation specifications of the class LightGroup in OCL are translated to Object-Z expressions (note that although we do not discuss a formalization of OCL in this paper, the example shows that translating OCL expressions to Object-Z is straightforward).



The dynamic behavior of the light group is defined using a statechart diagram (Fig. 11).

Formalizing the dynamic part of the system: The following Object-Z class *LightGroup* is an extension of the class presented above. Class operations are derived from

the statechart diagram in Fig. 11 according to the transformation rules described in the previous section.

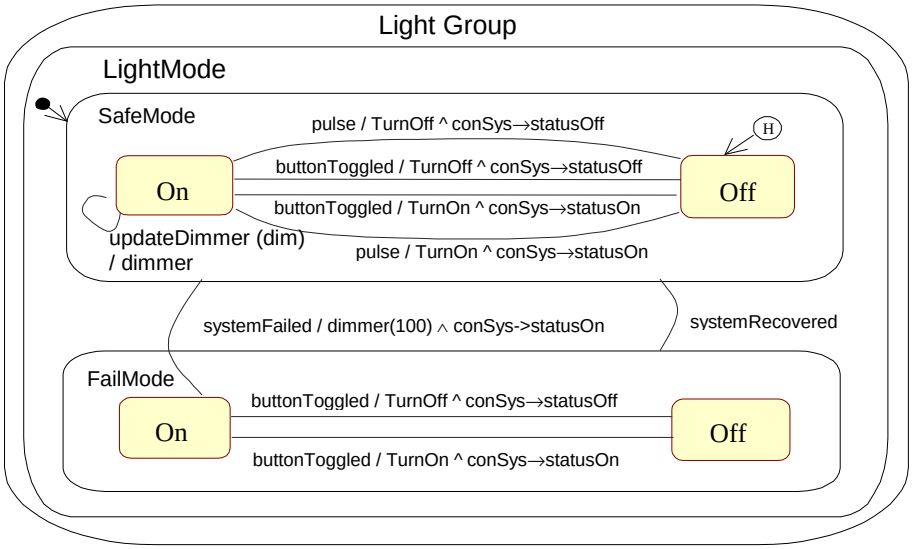


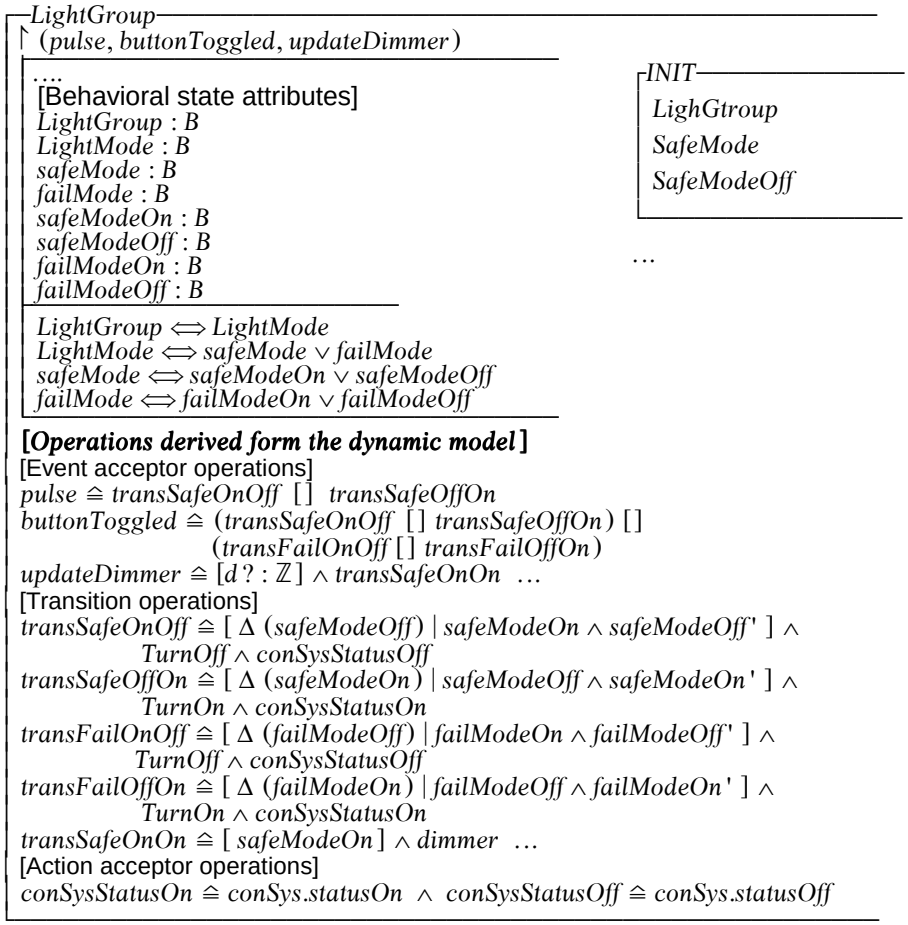
Figure 11. Statechart diagram for the light group

States are transformed as distinct behavioral state attributes and the state hierarchy is formalized as invariants. The initial state is formalized as the *Init* state schema of the Object-Z class.

The events, e.g. *updateDimmer*, *pulse*, and *buttonToggled* are transformed as distinct event acceptor operations (see the operations *updateDimmer*, *pulse*, and *buttonToggled*). The event acceptor operations invoke the transition operations defined for the transitions triggered by the events. Although the event *pulse* can trigger more than one transition, only one transition can fire at one time. For this reason, the operations corresponding to the transitions triggered by the event *pulse* are combined using the choice operator (see the operation *pulse*). This is the same for the event *buttonToggled*. The parameters of a call event *updateDimmer* are modeled as the input parameters of the Object-Z operation *updateDimmer* (see the operation *updateDimmer*).

All transitions are transformed into transition operations (see the operations *transSafeOnOff*, *transSafeOffOn*, and *transFailOnOff*). Behavioral state attributes corresponding to the source and target states of the transitions are defined as the pre or post-conditions of the transition operations. The transition operations also invoke the action operations corresponding to these effects. For call actions, e.g. *dimmer*, *TurnOn* and *TurnOff*, the Object-Z operations corresponding to these operations are combined as the action operations.

For the effect actions, e.g. *conSys→statusOff*, and *conSys→statusOn*, which propagate events [2] within other objects, e.g. the control system, the event acceptor operations corresponding to these events defined in the control system are combined using the dot (.) notation in Object-Z (see the operation *conSysStatusOn and conSysStatusOff*).



6. Conclusions

In this paper, we describe a formal Object-Z model of the UML metamodel and an extended metamodel for Object-Z. Given these two metamodels, we also present a formal transformation between the two languages at the meta-level. This metamodel-based transformation provides a precise, complete and consistent mapping between the two languages. With this integrated framework, transforming a UML model to Object-Z enables us to produce an analysis model that is unambiguous, verifiable and traceable. Also integrating both static and dynamic models in UML into a single Object-Z specification provides an integrated semantic basis for the semantic consistency check between the UML models. Examples of the semantic consistency checks include: checking that invariants are preserved, that no conflicts exist between invariants defined in the static model and guards defined in the dynamic models, and that no inconsistencies exist between object behaviors defined in two ways in terms of operation specifications and state machines. On the other hand, transforming an Object-Z

specification to UML enables us to visualize various aspects of the Object-Z specification. Although we do not discuss tools in this paper, existing tools for one language can be effectively used to help the analysis activity on models in the other language

References

[1] E. Börger and R. Gotzhein, Light control system: Problem description. 1999. http://rn.informatik.uni-kl.de/~recs/problem/.

[2] B. 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. Ledru, and M. Chabre-Peccoud. Translating the OMT Dynamic Model into Object-Z. in *ZUM'98- The Z Formal Specification Notation, 12th International Conference of Z users, LNCS. No. 1498,* Springer-Verlag. pp. 347-366, 1998.

[5] A. Evans and S. Kent, Core meta modelling semantics of UML: The pUML approach, *Proc. UML'99*, LNCS. No. 1723, pp. 140-155, 1999.

[6] R. France, J. Wu, M. M. Larrondo-Petrie, and J.-M. Bruel, A Tale of Two Case Studies: Using Integrated Methods to Support Rigorous Requirements Specification, *Proc. of the BCS FACS Methods Integration Workshop*, 1996.

[7] 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.

[8] K. Lano, *Formal Object-Oriented Development*, Springer 1995.

[9] W. McUmber and B. Cheng. A General Framework for Formalizing UML with Formal Languages. in *IEEE Conference on Software Engineering*. 2001.

[10] S-K. Kim and D. Carrington, Formalizing the UML class diagram using Object-Z, *Proc. UML'99*, LNCS, No. 1723, pp. 83-98, 1999.

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

[12] S-K. Kim and D. Carrington, A Formal Denotational Semantics of UML in Object-Z, *the special issue of the journal of l'Objet*, Vol. 7(1), pp. ??, 2001.

[13] S-K. Kim, D. Carrington, and R. Duke. A Metamodel-Based Transformation between UML and Object-Z. in *HCC'01 2001 IEEE Symposium on Visual Languages and Formal Methods*, IEEE Press. pp. 112-119, 2001.

[14] S-K. Kim and D. Carrington. A Formal Model of the UML Metamodel: the UML State Machine and its Integrity Constraints. in *Proc. of ZB 2002*. 2002, Springer. To Appear.

[15] J. Lilius and I. P. Paltor, Formalizing UML state machines for model checking, *Proc. UML'99*, LNCS, No. 1723, pp. 430-445, 1999.

[16] E. Meyer and J. Souquieres, A Systematic Approach to Transform OMT Diagrams to a B Specification, *FM'99*, Vol. 1, LNCS 1708, pp. 875-895, Springer-Verlag, 1999.

[17] OMG, *Unified Modeling Language Specification*, version 1.3, 1999, http://www.omg.org

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

[19] E. Wang, H. Richter and B. Chen, Formalizing and Integrating the Dynamic Model with OMT, *Proc. 19th International Conference on Software Engineering*, pp. 45 - 55, 1997.

[20] R. Wieringa, E. Dubois, and S. Huyts. Integrating Semi-formal and Formal Requirements. in *Advanced Information Systems Engineering, LNCS. No. 1250*, Springer. pp. 19-32, 1997.