An OCL-based Semantics of System State Modification Primitives

Andrés Vignaga

Instituto de Computación – Facultad de Ingeniería
Universidad de la República – Uruguay
avignaga@fing.edu.uy

Abstract

An object oriented system can be manipulated at a conceptual level by system state manipulation primitives. The possible manipulations include either queries or modifications to the state of the system. Modification primitives are used to describe the behavior of more complex state manipulations, such as system-level operations. A well defined semantics for these primitives is critical for a precise specification of system operations. In this paper, we present the semantics for the most common system state modification primitives. Their semantics will be specified by pre- and post-conditions expressed in OCL and using the UML metamodel as the model of systems for the OCL expressions. Every constraint was validated using the USE Tool.

Key Words: Semantics, System State Modification Primitives, OCL, UML, USE Tool

1 Introduction

The implementation of an object oriented system is rarely a direct mapping from its conceptual definition (i.e. a definition that takes full advantage of all the concepts of object technology). Rather, it is usually a representation of this conceptual definition restricted by the programming language or implementation technology used. For example, the concept of association is usually not supported by programming languages as a first class construct and is "something that needs to be designed". Also, compositions and association classes are eliminated during the design process for the same reason. Complex system state manipulations, such as system-level operations, are implemented as programs in terms of programming languages constructs driven by the representation of the system instead of its conceptual definition.

In turn, at a conceptual level, the state of a system can be manipulated by a number of operations that are not constructs of a programming language, rather they are conceptual operations that enable primitive queries and modifications to systems in their purest form and not in the technology driven representation. At this level, system operations are described in terms of these primitives operations. This

approach is more appropriate to reason about behavior since the primitives use pure object terminology and abstract away the actual representation of the system implementation. In fact, contracts as used in [2,3] aim at specifying the behavior of system operations by suggesting which primitives should apply during the execution of the operation. The primitives can be regarded as constructs for describing system state manipulation at a higher level of abstraction, and most importantly, independently of the designs needed to overcome the lack of some constructs present in most of the object oriented programming languages. In [9], a simple command language is given for modifying system states where the commands are precisely system state modification primitives.

The primitives seem to be the right tool to reason about and describe the behavior of system-level operations. Then a well defined semantics for them is crucial, both for writing and reading specifications, and for generating correct implementations. The problem addressed in this paper is the lack of a precise specification of the semantics of system state modification primitives.

Several authors [3,4,10] identified a set of five primitives ([10] referred to them as *system state manipulation commands*) which allow us to add and remove an instance from the system state, add and remove a link between instances from the system state, and update the value of an attribute. In some cases, the effect of each primitive is briefly explained in natural language [4,10] and in others [3] no description is given, and the semantics of each primitive must be guessed from the operation name. The UML Reference Manual [12] introduced the term *primitive* but described only some of them, namely those regarding object and link creation, and object destruction. Version 1.5 of UML [5] added to the UML Specification an Action Package [7] including a metamodel in which metaclasses represent "actions that a modeler can use". These actions include system state modification actions (e.g. `CreateObjectAction` metaclass), as well as others such as actions for querying a system. The metamodel and well-formedness rules clarify the syntax and action pre-conditions. However, the runtime effect of each action is again described in natural language. Finally, as mentioned above, [9] includes a command language where five of the commands correspond with each of the five primitives. Again, no semantics is given for those commands. Studying their behavior we found that one of them presents a flaw which will be pointed out in a later section.

In this paper, our main goal is to provide a precise semantics, yet informal, for system state modification primitives. We use the UML metamodel to represent both system state and the system static structure. Each primitive is represented by an operation, and its semantics is given by means of pre- and post-conditions expressed in OCL [8]. All OCL expressions were validated using the USE Tool [9].

This paper is organized as follows. Section 2 reviews system state modification primitives. Section 3 explains the details of the metamodel used. Section 4 introduces the semantics for each primitive. Section 5 contains a simple example of the effect of the primitives applied to a sample system state. Section 6 concludes.

2 System State Modification Primitives

We now review the set of five modification primitives and introduce a brief motivation for each of them. In order to achieve this and standarize terminology, we first review the concept of object oriented system, from both the static and the dynamic points of view.

2.1 System State and System Structure

A system is object oriented when it is organized at runtime as a collection of connected objects that incorporate data structure and behavior [11]. From this definition we may extract as first class components of that organization, which we shall call *system state*, the notions of *objects* and *connection between objects*. We can say that there is a general consensus in considering a system state as composed by:

- Objects (with values for its attributes)
- Links

A system may be described by one or more models [11]. An Object Model describes the static structure of objects in a system (their identity, their relationships to other objects, their attributes and their operations) [11]. We shall call this structure *system structure*, thus an Object Model describes the system structure, which in turn describes the set of all possible system states.

2.2 Primitives

Several authors [3,4,10] identified a set of five primitives, each of them allowing certain kind of modification upon a system state. According to the structure shown above, the possible modifications that are applicable to a system state are:

- **Object creation.** Instantiates a class existing in the system structure creating a new object and adding it to the system state.
- **Object destruction.** Removes an existing object from the system state.
- **Link creation.** Creates a connection from a tuple of objects and adds it to the system state.
- **Link destruction**. Removes a the connection between a tuple of objects from a system state.
- **Object attribute value update.** The value held by an object for one of its attributes is replaced by another value.

We shall name each primitive for further references. The following table maps each primitive to its name.

Primitive	Name
Object creation	create
Object destruction	destroy
Link creation	link
Link destruction	unlink
Object attribute value update	set

We model each primitive by an operation, and as said above, its semantics will be specified by pre- and post-conditions. In the next section we introduce the context for such constraints.

3 Metamodel for System State and System Structure

In order to be able to express in OCL pre- and post-conditions for the operations that represent the five primitives, we need an object oriented model that describes all the concepts considered relevant so far, that is, a metamodel for the system state and system structure.

3.1 Metamodel

The metamodel needed is just an object oriented representation of object oriented systems. For this reason, it seems quite reasonable that we use (part of) a well known existing model that fits our needs, instead of developing a new one. That model is the UML metamodel (level M2 of UML metamodeling architecture). A subset of the Core package together with a subset of the Common Behavior package and Data Types package will be enough for our purposes. Particularly, our metamodel includes metaclasses from Core - Backbone, Core – Relationships, and Core – Classifiers to model system structure, and Common Behavior – Instances and Common Behavior – Links to model system state. A graphical representation in the form of UML diagrams of the specified parts of the UML metamodel can be found in [6] in figures 2-5, 2-6 and 2-8 for the Core package, and in figures 2-16 and 2-17 for the Common Behavior package.
Besides the concepts that constitute the system state and the system structure, we need to represent the system state and system structure themselves. We add to our metamodel two classes, SystemState and SystemStructure representing those concepts respectively. These classes are associated representing an instance of the former class is an instance of (or conforms to) an instance of the latter class. We also add relationships for nesting every class of the Core package in class SystemStructure, and for nesting every class of the Common Behavior package in class SystemState. This approach is depicted in the (partial) class diagram shown in Fig. 1.

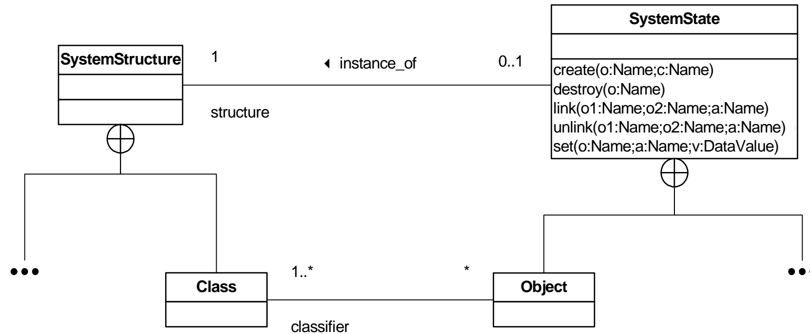

Figure 1: Structure of the metamodel

In this way, an object model describing a system structure (at level M1) in our metamodel is represented by a set of instances nested within an instance of class SystemStructure. In turn, an actual system configuration at runtime, that is, a system state (at level M0), is represented by a set of instances nested in an instance of class SystemState. Now, the state of a system can be treated as an instance with access to the actual contents of the state and its structure, and more importantly, we have a clear context for the operations representing the primitives.

Other approaches could have been used to structure our metamodel, concretely the parts concerning the state and its structure. The structure of the metamodel is conceptual, thus the alternative chosen is unimportant *per se*, as long as it fits our needs, that is, it gives us the ability to associate elements SystemStructure and SystemState, relate regular metaclasses to them, and also let SystemState own operations. For example, modeling both SystemState and SystemStructure using subsystems could have been appropriate.

Finally, our metamodel supports multiple classification, but we do not provide a primitive to add or remove classes to objects such as action ReclassifyObjectAction in [7]. Thus, every object has exactly one class (the class used for instantiation when the object was created). For example, in the context of an object, the expression self.classifier is still of type Set(Classifier) but always yields a set with one element. This feature is widely used in the semantics in the next section.

3.2 Well-formedness

An important issue we did not discuss so far is state well-formedness. A model (in particular a state) is said to be well-formed if it satisfies all predefined and model-specified rules or constraints [12]. Predefined rules and constraints are those imposed by the UML metamodel (e.g. an attribute may not realize a node), and we assume them all satisfied. Model-specified rules and constraints include the structure of the system (such as whether a class or an association exists and thus can be used, and association multiplicities), and system invariants. System

invariants go well beyond the scope of this work, so they are not considered. According to our semantics, structural well-formedness is achieved by construction, except for multiplicities. That is, we do not impose ourselves the invariant that system states are well-formed respect to multiplicities. So, the states we handle can be either well-formed or ill-formed. Well-formedness of a state can be checked anytime. As a consequence, we do not prevent the execution of a primitive that yields an ill-formed state. The reason for this is that it is often common that the path (sometimes all the paths) that leads to a certain well-formed state requires passing through intermediate states that are ill-formed [12]. Thus, restricting ourselves to only well-formed states is not an appropriate approach. In conclusion, the primitives may yield to states that are ill-formed respect to multiplicities; well-formedness might be checked separately, if desired.

3.3 Sample System Structure

We now introduce a sample object model and its correspondent as an instance of the metamodel defined above which will be used in the case study in section 5. The object model shown in Fig. 2 has been chosen to be as simple as possible. In Fig. 3 we show an instance of our metamodel representing the model in Fig. 2. We elided the instance of class SystemStructure to avoid introducing noisy links in the diagram. Statically, there is no system running, thus there is no instance of class SystemState.

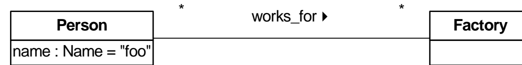

Figure 2: Sample object model

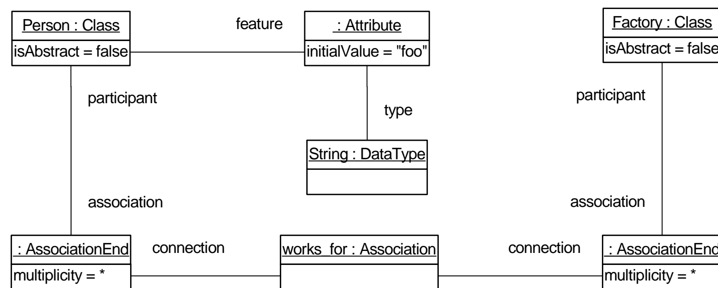

Figure 3: Instance of the metamodel

4 Semantics for Modification Primitives

In this section we present the specification of system state modification primitives. Based on the metamodel introduced in the previous section, in the following subsections we show OCL expressions that contain pre- and post-conditions for the operations specified for the SystemState class. The semantics of the primitives is specified by those expressions. Each OCL expression is preceded by a description of the primitive, in which we dive into its details and which could be useful for the reader to understand the OCL code. It could be also convenient for the reader to have at hand a copy of the diagrams of the UML metamodel mentioned in subsection 3.1. The last subsection includes all the additional operations needed for the specification.

As a final remark, the reader should notice that all the types used in this specification were taken from the UML metamodel, including those for the parameters of the operations modeling the primitives. As shown in Fig. 1, we used names (instances of Name) for identifying objects, classes, associations and so on in the system state or system structure, and data values (instances of DataValue) for referring to values of any type.

4.1 Semantics for create

This primitive lets us instantiate a class (i.e. create a new instance) and add the resulting instance to the system state. It needs a unique name for the instance to be created, as well as the name of the class to be instantiated. Thus, the chosen name for the instance might not have been used for an instance already in the system state, and the class from whom that instance will be created must be an existing class in the system structure, and it might not be abstract. After the primitive is completed, the system state owns a new instance of the specified class.

The UML Action Semantics [7] specifies that after the execution of an instance of the CreateObjectAction class no attribute values are set for the created instance (the default values for its attributes), that is, no constructor executes. The UML Reference Manual [12] is consistent with this approach, but distinguishes two stages in the actual creation of an object; first, the allocation of the new instance in the environment of the system (instantiation), and second, the initialization of its attributes (initialization), which occurs immediately after the instantiation. It also warns that an object instantiated but not initialized (called raw instance) might be inconsistent and “is not available to the rest of the system until it has been initialized”. We expect an instance to be available after creation, so for practical reasons, we decided to assume that the create primitive should both instantiate a class producing an instance, and initialize it with the default values found in its descriptor.

Next is the OCL expression that specifies the create primitive.

```
context SystemState::create(o:Name,c:Name)

pre: -- Exists a class named "c" in the associated system structure and
     -- is not an abstract class
     self.structure.ownedInstance->exists(e | e.oclIsTypeOf(Class) and
     e.name = c and e.isAbstract = false)

pre: -- There is no object with name "o" in the system state
     not self.ownedObject->exists(e | e.name = o)

post: -- Exists a new object of class named "c" that is named "o" and all
      -- its attributes are initialized with their default values

     -- C is the set of descriptors needed to create the new object
     let C = self.structure.ownedInstance->select(e | e.oclIsTypeOf(Class)
        and e.name = c)->any(true).oclAsType(Class).getAncestors() in
     -- O is the new object
     let O = self.ownedObject->select(e | e.name = o and
        e.oclIsNew())->any(true) in
           -- C is the classifier for O
           O.classifier->includes(C) and
           -- the new instance is owned by self
           O.owner = self and
           -- self is the owner of the new instance
           self.ownedObject->includes(O) and
           -- for every feature a of class C
           C.feature->forAll(a | a.oclIsTypeOf(Attribute) implies
              -- if feature a is an attribute implies that a is attached
              -- to an attribute link al that
              a.oclAsType(Attribute).attributeLink->exists(al |
                 -- is new
                 al.oclIsNew() and
                 -- is attached to the new object O
                 al.instance = O and
                 -- its value is the default value
                 al.value.oclAsType(DataValue).value =
                    a.oclAsType(Attribute).initialValue and
                 -- is owned by the system state
                 al.owner = self and
                 self.ownedAttributeLink->includes(al)
              )
           )
```

4.2 Semantics for destroy

This primitive can remove an instance from the system state. It only needs the name of the object to be removed, and requires that an object with that name exists in the system.

After the primitive is executed, the specified object is no longer available in the system. This implies that any link that involved that instance is also removed from the system state. Moreover, any composite object should be recursively removed.

In our experience using the USE Tool, we detected that the command corresponding to this primitive did not work properly in all cases. In fact, the specified object and all the involved links are actually removed, but the composing objects (if exist) are not, and they must be removed explicitly.

Next is the OCL expression that specifies the destroy primitive.

```
context SystemState::destroy(o:Name)

pre: -- Exists an object with name "o"
     self.ownedObject->exists(e | e.name = o)

post:
     let O = self.ownedObject@pre->select(e | e.name@pre = o)->any(true) in
     let C = O.classifier@pre->any(true) in
     let cs = C.oclAsType(Class).getTree() in
     let insts:Sequence(Instance) = cs->iterate(
        x:Class;acc:Sequence(Instance)=Sequence{} |
        acc->union(x.instance@pre->asSequence)
     ) in
     -- The set of all objects removed (including composites)
     let all:Set(Instance) = insts->iterate(
        e:Instance;acc:Set(Instance)=Set{O} |
        if e@pre.linkEnd@pre->collect(link@pre)->asSet->
           collect(connection@pre)->asSet->flatten->collect(instance@pre)->
           asSet->excluding(e)->intersection(acc)->size()>0 then
              acc->including(e)
        else
              acc
        endif
     ) in
     -- The set of all links removed
     let ls:Set(Link) = self.ownedLink@pre->select(e | e.connection@pre->
        exists(le:LinkEnd | not le.instance@pre->intersection(all)->
        isEmpty())
     ) in
     -- The set of all link ends removed
     let les:Set(LinkEnd) = self.ownedLinkEnd@pre->select(e |
        ls->includes(e.link@pre)
     ) in
        -- All the objects were removed together with their attribute links
        all->forAll(z | self.ownedObject->excludes(z)
           and self.ownedAttributeLink->excludesAll(z.slot@pre)
        ) and
        cs->forAll(instance->excludesAll(all)) and
        -- No attribute is connected to any of the removed objects
        all->forAll(y | test->forAll(feature->forAll(f |
           f.oclIsTypeOf(Attribute) implies
           f.oclAsType(Attribute).attributeLink->select(instance = y)->
           isEmpty()))
        ) and
        -- All the links and link ends were removed
        self.ownedLink->excludesAll(ls) and
        self.ownedLinkEnd->excludesAll(les)
```

4.3 Semantics for link

This primitive lets us establish a connection between a tuple of objects. In this paper we restrict ourselves to binary associations, thus this tuple is actually a pair of objects. We also do not consider association classes, since these do not introduce interesting variants, as explained later. Thus every association is a direct instance of Association metaclass.
The ability to connect the pair of objects must have been declared in the system structure, thus there must be an association between the classes of objects (or any of its ancestors) we want to connect. For the execution of this primitive to take place it is required that the objects to connect exist in the system state, and since the extent of an association (i.e. the collection of connections between instances of the associated classes) is a set, the objects might not be already connected.
After the primitive is completed, there is a new link between the pair of specified objects according to the specified association.
Considering the case of association classes would require the creation of an instance of metaclass LinkObject instead of Object. Since it is a Link, the semantics shown above still holds, and since it is an Object it is also needed to initialize its attributes as we did with create, using property feature of AssociationClass.
Next is the OCL expression that specifies the link primitive.

```
context SystemState::link(o1:Name,o2:Name,a:Name)

pre: -- Objects named o1 and o2 exists in the system state
     self.ownedObject->exists(e | e.name = o1) and
     self.ownedObject->exists(e | e.name = o2)

pre: -- Exists an association named "a" in the associated object model
     self.structure.ownedInstance->exists(e | e.oclIsTypeOf(Association) and
        e.name = a)

pre: -- Classes of objects named "o1" and "o2" must be connected by
     -- association named a
     let O1 = self.ownedObject->select(e | e.name = o1)->any(true) in
     let O2 = self.ownedObject->select(e | e.name = o2)->any(true) in
     let A = self.structure.ownedInstance->select(e |
        e.oclIsTypeOf(Association) and e.name = a)->
        any(true)->oclAsType(Association) in
     let c1 = O1.classifier->any(true) in
     let c2 = O2.classifier->any(true) in
        c1.allParents()->including(c1)->intersection(A.connection.type->
           asSet)->size() = 1 and
        c2.allParents()->including(c2)->intersection(A.connection.type->
           asSet)->size() = 1

pre: -- Objects named "o1" and "o2" are not already linked by "a"
     let O1 = self.ownedObject->select(e | e.name = o1)->any(true) in
     let O2 = self.ownedObject->select(e | e.name = o2)->any(true) in
     let A = self.structure.ownedInstance->select(e |
        e.oclIsTypeOf(Association) and e.name = a)->
        any(true)->oclAsType(Association) in
           A.link->select(l | l.connection.instance->includes(O1) and
           l.connection.instance->includes(O2))->isEmpty()
```

```
post: -- There is a new link (instance of association named "a") between
       -- objects named o1 and o2
     let O1 = self.ownedObject->select(e | e.name = o1)->any(true) in
     let O2 = self.ownedObject->select(e | e.name = o2)->any(true) in
     let A = self.structure.ownedInstance->select(e |
        e.oclIsTypeOf(Association) and e.name = a)->
        any(true)->oclAsType(Association) in
     let c1 = O1.classifier->any(true) in
     let c2 = O2.classifier->any(true) in
     let C1 = c1.allParents()->including(c1)->
        select(w | w.oclAsType(Class).associationEnd.association->
        includes(A))->any(true) in
     let C2 = c2.allParents()->including(c2)->
        select(w | w.oclAsType(Class).associationEnd.association->
        includes(A))->any(true) in
     -- There is a new link l that
     let l = self.ownedLink->select(e | e.oclIsNew())->any(true) in
        -- is attached to association A
        l.association = A and
        -- has two link ends
        l.connection->size() = 2 and
        -- one of them is new and is attached to object O1 and to A's
        -- association end at C1's end
        l.connection->exists(le:LinkEnd | le.oclIsNew() and
           C1=le.associationEnd.type and le.associationEnd.association = A
           and le.instance = O1 and le.owner = self and self.ownedLinkEnd->
           includes(le)
        ) and
        -- the other is new too and is attached to object O2 and to A's
        -- association end at C2's end
        l.connection->exists(le:LinkEnd | le.oclIsNew() and
           C2=le.associationEnd.type and le.associationEnd.association = A
           and le.instance = O2 and le.owner = self and self.ownedLinkEnd->
           includes(le)
        )
```

4.4 Semantics for unlink

Using this primitive a link among instances can be removed. As said in the previous subsection we concern ourselves only with ordinary associations. The case of association classes is discussed later.

This primitive requires that the specified association exists in the system structure and associates the classes of the specified objects (or any of its ancestors). It is also required that the objects to disconnect exist in the system state and are already linked specifically by the specified association.

After completion, the link between the specified pair of objects through the specified association is removed.

The case of an association class would require removing an instance of metaclass LinkObject. That instance is both a Link and an Object. The semantics below still holds for the “link” part of the instance. For the “object” part, instance destruction (i.e. attribute destruction, and possibly link and composite destruction) apply as in destroy.

Next is the OCL expression that specifies the unlink primitive.

```
context SystemState::unlink(o1:Name,o2:Name,a:Name)

pre: -- Objects named "o1" and "o2" exists in the system state
     self.ownedObject->exists(e | e.name = o1) and self.ownedObject->
     exists(e | e.name = o2)
pre: -- Exists an association named "a" in the associated object model
     self.structure.ownedInstance->exists(e | e.oclIsTypeOf(Association) and
     e.name = a)

pre: -- Classes of objects named "o1" and "o2" must be connected by
     -- association named "a"
     let O1 = self.ownedObject->select(e | e.name = o1)->any(true) in
     let O2 = self.ownedObject->select(e | e.name = o2)->any(true) in
     let C1 = O1.classifier->any(true) in
     let C2 = O2.classifier->any(true) in
     let A = self.structure.ownedInstance->select(e |
        e.oclIsTypeOf(Association) and e.name = a)->any(true)->
        oclAsType(Association) in
           C1.allParents()->including(C1)->intersection(A.connection.type->
           asSet)->size() = 1 and
           C2.allParents()->including(C2)->intersection(A.connection.type->
           asSet)->size() = 1

pre: -- Objects named "o1" and "o2" are linked by association named "a"
     let O1 = self.ownedObject->select(e | e.name = o1)->any(true) in
     let O2 = self.ownedObject->select(e | e.name = o2)->any(true) in
     let A = self.structure.ownedInstance->select(e |
        e.oclIsTypeOf(Association) and e.name = a)->any(true)->
        oclAsType(Association) in
           not A.link->select(l | l.connection.instance->includes(O1) and
           l.connection.instance->includes(O2))->isEmpty()

post: -- The link between objects named "o1" and "o2" by association named
      -- "a" is no longer available in the system state
     let O1 = self.ownedObject->select(e | e.name = o1)->any(true) in
     let O2 = self.ownedObject->select(e | e.name = o2)->any(true) in
     let A = self.structure.ownedInstance->select(e |
        e.oclIsTypeOf(Association) and e.name = a)->any(true)->
        oclAsType(Association) in
     let c1 = O1.classifier->any(true) in
     let c2 = O2.classifier->any(true) in
     -- The class of O1 at A's end
     let C1 = c1.allParents()->including(c1)->
        select(w | w.oclAsType(Class).associationEnd.association->
        includes(A))->any(true) in
     -- The class of O2 at A's end
     let C2 = c2.allParents()->including(c2)->
        select(w | w.oclAsType(Class).associationEnd.association->
        includes(A))->any(true) in
     -- The link end that connected O1 to the link removed
     let le1:LinkEnd = O1@pre.linkEnd@pre->select(e |
        e@pre.link@pre.association@pre = A and
        e@pre.link@pre.connection@pre->exists(instance@pre = O2)
        )->any(true) in
     -- The link end that connected O2 to the link removed
     let le2:LinkEnd = O2@pre.linkEnd@pre->select(e |
        e@pre.link@pre.association@pre = A and
        e@pre.link@pre.connection@pre->exists(instance@pre = O1)
        )->any(true) in
     -- The link removed
     let l:Link = le1.link@pre in
```

```
-- Both link ends are not accessible from the disconnected objects
O1.linkEnd->excludes(le1) and
O2.linkEnd->excludes(le2) and
-- also not accessible from any association end
C1.oclAsType(Class).associationEnd->forAll(linkEnd->excludes(le1))
and
C2.oclAsType(Class).associationEnd->forAll(linkEnd->excludes(le2))
and
-- they are not available in the system state
self.ownedLinkEnd->excludes(le1) and
self.ownedLinkEnd->excludes(le2) and
-- The link is not accessible from the association
A.link->excludes(l) and
-- it is not available in the system state
self.ownedLink->excludes(l)
```

4.5 Semantics for set

Using this primitive the value of an attribute of an object can be changed. It takes as input the target object, the attribute, and the new value for it. It is required that the target object exists in the system state, and the specified attribute is defined in the target object class (or any of its ancestors). Also, the type of the new value must conform to the type of the attribute (i.e. either both types match or the type of the value is a subtype of the type of the attribute).
After the primitive is completed, the object holds the new value for the attribute.
Next is the OCL expression that specifies the set primitive.

```
context SystemState::set(o:Name,a:Name,v:DataValue)

pre: -- Object named "o" exists in the system state
     self.ownedObject->exists(e | e.name = o)

pre: -- The class of object named "o" has an attribute named "a"
     let O:Object = self.ownedObject->select(e | e.name = o)->any(true) in
     let C:Classifier = O.classifier->any(true) in
        C.allParents()->including(C)->exists(oclAsType(Class).feature->
           exists(e | e.oclIsTypeOf(Attribute) and e.name = a))

pre: -- The type of value v mathces the type attribute named a of the class
     -- of object named o
     let c = self.ownedObject->select(e | e.name = o)->any(true).classifier
        ->any(true) in
     let C = c.allParents()->including(c) in
     let t = C->collect(oclAsType(Class).feature)->flatten->select(e |
        e.oclIsTypeOf(Attribute) and e.name = a)->
        any(true).oclAsType(Attribute).type in
           v.classifier->collect(allParents())->flatten()->
           union(v.classifier)->includes(t)

post: -- The value for attribute named a in object named o is now v
     let O:Object = self.ownedObject->select(e | e.name = o)->any(true) in
     let A:Attribute = self.structure.ownedInstance->select(e |
        e.oclIsTypeOf(Attribute) and e.name = a)->any(true)->
        oclAsType(Attribute) in
     let al:AttributeLink = O.slot->intersection(A.attributeLink)->
        any(true) in
           al.value = v
```

4.6 Additional Operations

We now show all the additional operations used in the specifications organized by the metaclass where they are intended to appear.

<u>GeneralizableElement</u>

The operation parents(), borrowed from [6], returns a Set containing all the Generalizable Elements directly inherited by this GeneralizableElement, excluding itself:

```
parent() : Set(GeneralizableElement) =
   self.generalization->collect(g | g.parent)->asSet
```

The operation allParents(), also borrowed from [6], returns a Set containing all the Generalizable Elements inherited by this GeneralizableElement (the transitive closure), excluding the GeneralizableElement itself:

```
allParents() : Set(GeneralizableElement) =
   self.parent()->union(
      self.parent()->collect(g | g.allParents())->flatten)->asSet
```

The operation offspring() returns a Set containing all the Generalizable Elements that are direct descendants from this GeneralizableElement, excluding itself:

```
offspring() : Set(GeneralizableElement) =
   self.specialization->collect(g | g.child)->asSet
```

The operation allOffspring() returns a Set containing all the Generalizable Elements that are descendants from this GeneralizableElement, excluding itself:

```
allOffspring() : Set(GeneralizableElement) =
   self.offspring()->union(
      self.offspring()->collect(g | g.allOffspring())->flatten)->asset
```

<u>Class</u>

The operation getComponents() returns a Sequence containing all the classes related by composition with this Class and all its descendants (the transitive closure), ordered in a depth first search (DFS) basis:

```
getComponents():Sequence(Class) =
  self.associationEnd->iterate(
     x:AssociationEnd;
     acc:Sequence(Class)=Sequence{self}->union(
        self.allOffspring()->iterate(
           y:GeneralizableElement;
           acc2:Sequence(Class)=Sequence{}
           | acc2->including(y.oclAsType(Class))))
```

```
  | if x.aggregation = #composite then
       acc->union(x.association.connection->asSet->excluding(x)->
          any(true).type.oclAsType(Class).getComponents())
    else
       acc
    endif
)
```

5 Examples

In this section we show an example of the application of each primitive to an instance of the sample model introduced in section 3. The examples consist of a pair of snapshots of our system, one that satisfies the pre-condition, and the other satisfying the post-condition of each primitive. This can be regarded as an animation of the system showing graphically the effect of the primitives.
We assume that when the system is running, an instance named ss of class SystemState is available, and linked to the instance of SystemStructure, as shown in Fig. 4 (only showing one of the links between the structure and the actual instances that compose the system structure). Except for the first example, for clarity we elided in all the snapshots the instances of classes SystemState and SystemStructure and all the links involving them.

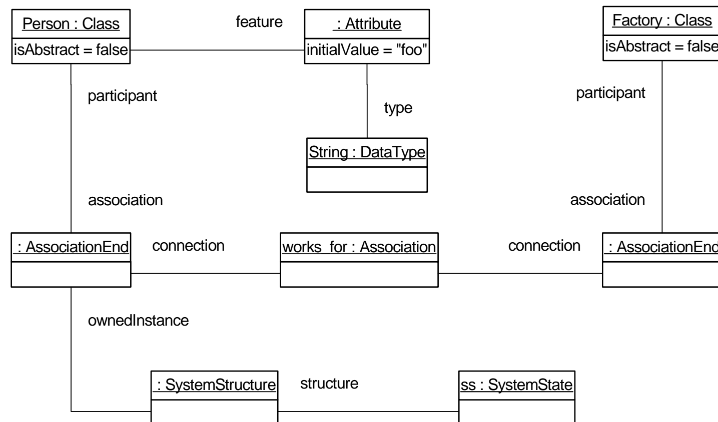

Figure 4: Snapshot of an empty system state

5.1 Example of create

For this primitive we can use Fig. 4 as a snapshot that satisfies the pre-conditions. It represents the empty state, since no object is present.
After the invocation of ss.create("p","Person") a new person was created and its attributes (in our case attribute name) are initialized with the default values. A snapshot of the resulting state that satisfies the post-condition is shown in Fig. 5.

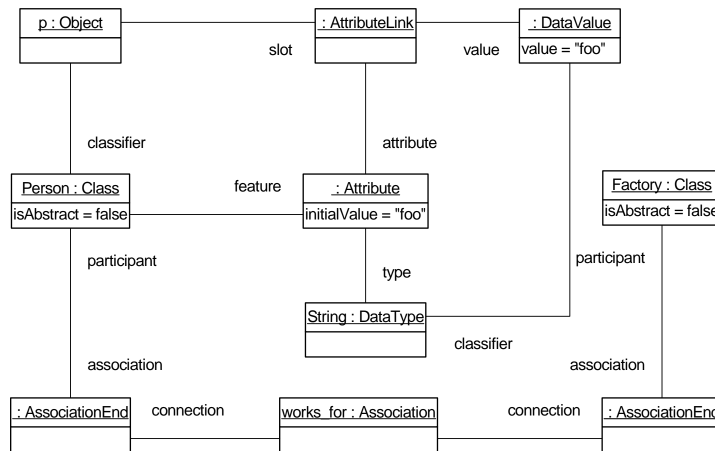

Figure 5: Snapshot after create

5.2 Example of destroy

We can use the same pair of snapshots to illustrate the effect of the destroy primitive. Snapshot in Fig. 5 satisfies the pre-conditions. After the invocation of ss.destroy("p") the state is empty again, and the result is shown in Fig. 4.

5.3 Example of link

To apply this primitive we need at least two instances, so we create a person and a factory invoking ss.create("p","Person") and ss.create("f","Factory") to the empty state. The result is shown in Fig. 6 and that snapshot satisfies the pre-conditions of link.

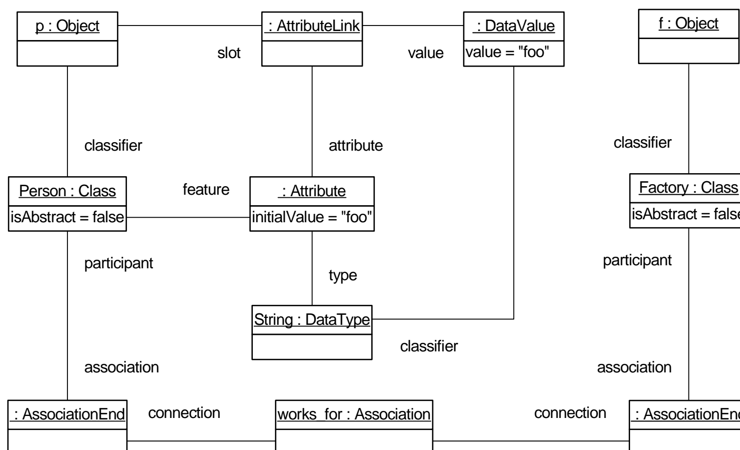

Figure 6: Snapshot before link

After the invocation to ss.link("p","f","works_for") there is a new link that connects objects p and f via its respective link ends. The resulting state is shown in Fig. 7.

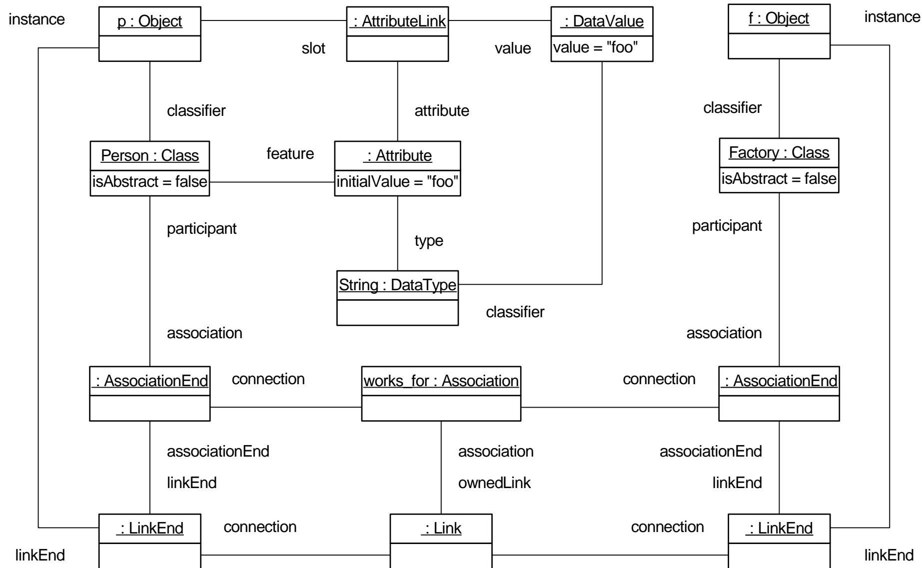

Figure 7: Snapshot after link and before unlink

5.4 Example of unlink

Again we use the same pair of snapshots of the previous subsection to illustrate the effect of unlink. Snapshot of Fig. 7 is valid before the call ss.unlink("p","f","works_for"). After the invocation, the two objects are disconnected since the link and link ends were removed. The result is again the snapshot in Fig. 6.

5.5 Example of set

For this primitive, we start from the state of Fig. 4 where only object p exists. For the invocation ss.set("p","name",dv) we need the data value dv shown in Fig. 8.

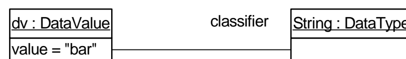

Figure 8: Data value for applying set

After the invocation, the value of attribute name of p is dv. The result is shown in Fig. 9.

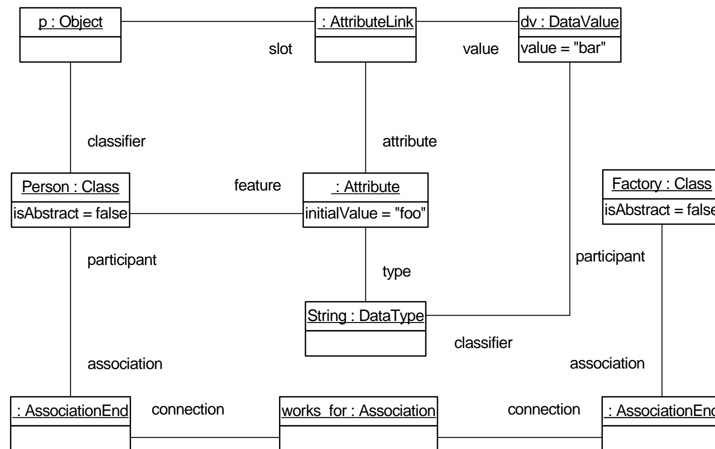

Figure 9: Snapshot after set

6 Conclusions

We have written a specification of the semantics of system state modification primitives. Although this specification is not completely formal, it is more rigorous than other versions found in the bibliography. We used the UML metamodel as a description of both system structure and system state, and OCL for expressing the pre- and post-conditions that specify the semantics of each primitive. All OCL code was validated using the USE Tool.

We used for validation the UML metamodel codification that comes as an example with the USE installation. Some minor changes were introduced to it: (meta) classes SystemState and SystemStructure, relationships between them and other metaclasses, and the additional operations of subsection 4.6. The main addition was the OCL expressions of section 4. We also generated more exhaustive test cases than those shown in section 5. It took about 30 man-hours to write the specification and complete the validation process. The specification file together with the test scripts is available at `www.fing.edu.uy/~avignaga/Primitives`.

A semantics like the one we introduced in this work can be used to understand the effect of the primitives both when reasoning about the behavior of a system-level operation in order to write a precise specification for it, and when reading an existing specification, especially when designing or implementing a program that satisfies that specification. It can also be applied in the same manner to other contexts, such as subsystems, and even to components, since the semantics of component operations is a vital part of the component interface specification [1]. Moreover, it can be applied to operations of any interface realized by a coarse grained class, such as SystemState.

Finally, it could be a start point towards a formal specification of the semantics of the primitives. A formal specification could be used in a formal definition of a

language such as provided by [9]. Most importantly, it could provide a framework for formally reasoning about the use of primitives, which in combination with a proof assistant would make it possible to prove certain kind of properties on system states.

References

[1] J. Cheesman and J. Daniels. *UML Components: A simple process for specifying Component-Based Software.* Addison-Wesley, 2001.

[2] D. D'Souza and A. Wills. *Objects, Components and Frameworks with UML: The Catalysis Approach.* Addison-Wesley, 1998.

[3] C. Larman. *Applying UML and Patterns.* Prentice Hall, second edition, 2002.

[4] I. Oliver. *'Executing' the OCL.* In ECOOP'99 Workshop for PhD Students in OO Systems (PhDOOS '99), 1999.

[5] OMG. *OMG Unified Modeling Language Specification, Version 1.5, March 2003.* Object Management Group, Inc., Framingham, Mass., Internet: `http://www.omg.org`, 2003.

[6] OMG. UML Semantics. In *OMG Unified Modeling Language Specification, Version 1.5, March 2003* [3], chapter 2.

[7] OMG. UML Action Semantics. In *OMG Unified Modeling Language Specification, Version 1.5, March 2003* [3], sections 2.14-2.25.

[8] OMG. Object Constraint Language Specification. In *OMG Unified Modeling Language Specification, Version 1.5, March 2003* [3], chapter 6.

[9] M. Richters. The USE Tool: A UML-based specification environment, 2001. Internet: `http://www.db.informatik.uni-bremen.de/projects/USE/`.

[10] M. Richters. *A precise approach to validating UML models and OCL constraints.* Biss Monographs, Logos Verlag Berlin, 2002.

[11] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen. *Object-Oriented Analysis and Design.* Prentice Hall, Engelwood Cliffs (NJ), 1991.

[12] J. Rumbaugh, I. Jacobson and G. Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1998.