Some Shortcomings of OCL, the Object Constraint Language of UML

Mandana Vaziri and Daniel Jackson
MIT Laboratory for Computer science

December 7, 1999

1 Introduction

The purpose of this paper is to expose some shortcomings of the Object Constraint Language (OCL). We argue that, despite its numerous benefits, OCL is too implementation-oriented and therefore not well-suited for conceptual modelling. Moreover, it is at times unnecessarily verbose, yet far from natural language.

In the past couple of years, we have designed a language, Alloy, that has similar motivations to OCL, while attempting to avoid some of these drawbacks. To illustrate the weaknesses of OCL, and some ways in which it may be improved, we have translated most of the core package of the UML metamodel, together with its well-formedness constraints, into Alloy. Moreover, we have shown that this subset of the UML metamodel is consistent, provided that this translation is valid, using Alloy's analysis tool, Alcoa.

Section 2 describes the requirements of OCL and its shortcomings with respect to these requirements. Section 3 gives an overview of Alloy. Section 4 presents the translation of the UML metamodel into Alloy, and illustrates the issues raised in Section 2. Finally, Appendix A contains the complete translation of the metamodel classes into Alloy, and Appendix B the translation of the metamodel constraints.

2 Shortcomings of OCL

The OCL manual [1] describes the requirements of OCL as follows:

1. *OCL must be able to express extra (necessary) information on the models and other artifacts used in object-oriented development.*

2. *OCL must be a precise, unambiguous language that can easily be read and written by all practitioners of object technology and by their customers. This means that the language must be understood by people who are not mathematicians or computer scientists.*

3. *OCL must be a declarative language. Its expression can have no side-effects; that is, the state of a system must not change because of an OCL expression. [...]*

4. *OCL must be a typed language. [...]*

Despite its numerous benefits, we find that OCL has the following shortcomings with respect to these requirements. For each concern, Section 4 provides an illustration using the UML metamodel.

- OCL seems to be closer to an implementation language than to a conceptual language, because it uses operations in constraints, and its type system is close to that of an object-oriented programming language.
  - The use of operations in constraints appears to be problematic in two respects.
    - First, this adds to the operational flavor of OCL. An operation may go into an infinite loop or be undefined. In these cases, the expression containing the operation is said to be undefined. This adds unnecessary complexity to the language and raises concerns about its precision such as: What does it mean for a model to satisfy an undefined constraint? How do we know that a constraint is undefined?
    - Secondly, consider an operation applied to a collection of objects that are instances of some class. A subset of these objects may also be instances of a subclass of that class, and the subclass may redefine the operation. It is not clear what operation gets applied to each object in the collection. Furthermore, even if we assume that we apply the redefinition of the operation to objects which are instances of the subclass, then this implies that the meaning of a constraint may change as the model evolves and subclasses are added. This is quite undesirable, since the expression of the constraints of a system may have to be revised as the model evolves.
  - OCL's type system is unnecessarily complicated. Conceptually, a class is a set of objects, and a subclass is a subset of these objects. So if a class inherits from two classes, then the two classes must be non-disjoint sets. In OCL, it is possible to have a class which inherits from two seemingly disjoint classes.
- OCL expressions are at times unnecessarily verbose due to the following:
  - OCL uses two different symbols to navigate through sets and scalars (`->` and `.` respectively). This lack of uniformity adds unnecessary complexity and makes OCL expressions less succinct.
  - Classes are not treated simply as collections of objects. As a result, we cannot use set operators to manipulate them directly, which increases the use of quantifiers. This, in addition to the implementation-based type system, makes frequent coercions (`oclIsKindOf`) necessary.
- OCL expressions are often unnecessarily hard to read:
  - Subexpressions are often not textually placed where they normally would be in English, because quantifiers (`forall`), and collection operators are "stacked", i.e. followed through via navigation. This also makes parenthesis matching difficult.
  - Logical operators (`not`,...) are not stacked, and this causes the separation of certain logical and set operators, making some expressions hard to read.
- Finally, OCL is not a stand-alone language: a model always needs an acompanying UML class diagram. There are many advantages to having a stand-alone constraint language, including the following:

- The choice between expressing models graphically or textually can be made more flexibly.
- There is better integration between the modelling language and the constraint language.
- The constraint language's semantics is easier to define.
- The constraint language is more amenable to automated analysis.

3 Description of the Alloy modelling language

Alloy [3] is a simple, precise, and tractable notation for object modelling. It is amenable to automatic analysis, and its analysis tool, Alcoa [5], can perform simulations, as well as consistency checking. Alloy also has a graphical sublanguage, which provides the user with the flexibility to express specifications graphically or textually.

Alloy's simplicity lies in the following design choices. Scalars are treated as singleton sets; this allows a uniform treatment of sets and scalars during navigation. Types are implicit and are associated with *domains*. A domain is just a set that is not declared as a subset of any other set. Classes correspond to sets that are subsets of the domains, and subclasses corresponds to subsets.

An Alloy specification consists of several *paragraphs*. The *domain paragraph* declares all domains used in the specification. The *state paragraph* declares additional sets whose elements are drawn from the domains, and relations between sets. Declarations may include multiplicity constraints, mutability constraints, and may specify that the subsets of a set are disjoint or form a partition. Following these paragraphs, the specification includes invariants on the state.

The following is a small example of an Alloy specification, defining domains `Person` and `Name`. The sets `Man` and `Woman` partition `Person` "statically", i.e. a man cannot become a woman and vice versa. The relation `parents` is declared to be "right-static", i.e. a person's parents do not change. The relation `siblings` is many to many, while `wife` relates at most one man to at most one woman, `name` relates a person to a unique name.

The paragraph starting with `def` defines `siblings` as a derived relation, and the `inv` paragraph gives some basic invariants. The keywords `all`, `some`, `no`, and `sole` denote quantifiers, where `sole` means at most one. The symbols `in`, `&`, and `+` denote set inclusion, intersection, and transitive closure respectively. Finally, `&&`, and `||` are logical and, and or, respectively. The symbol `//` indicates a comment.

```
model Family{
 domain{Person, Name}
 state{
   partition Man, Woman: static Person
   Married: Person
   parents: Person -> static Person
   siblings: Person -> Person
   wife(~husband): Man? -> Woman?
   name: Person -> Name!
 }
 def siblings {
   all a,b | a in b.siblings <-> (a.parents = b.parents)
```

```
  //Two persons are siblings if and only if they have the same parents
 }
 inv Basics{
   all p | some p.wife <-> p in Man & Married
   //A person has a wife if and only if he's a man and is married.

   no p | p.wife in p.siblings
   //No person's wife is also a sibling

   all p | (sole p.parents & Man) && (sole p.parents & Woman)
   //A person has at most one father and at most one mother.

   no p | p in p.+parents
   //No person is an ancestor of him/herself.
 }
}
```

This example is explained in more detail and compared to an OCL version in [4].

4 The UML metamodel

In this section, we illustrate the issues presented in Section 2 using examples from the UML metamodel [2].

OCL constraints and operations are local to a particular class. We indicate that by preceding the constraints or operations with the name of the class to which they apply, using the format of [1]. This is not needed for Alloy, since invariants in Alloy are global.

- Our first series of examples illustrates the problems with having operations in constraints. Consider the definition of the `allParents` operation in OCL:

OCL	`GeneralizableElement`
	`allParents: Set(GeneralizableElement);`
	`allParents = self.parent->union(self.parent.allparents)`

Alloy	`all e: GeneralizableElement |` `e.allParents = e.+parent`
English	The operation `allParents` returns a set containing all the GeneralizableElements inherited by this GeneralizableElement (the transitive closure), excluding the GeneralizableElement itself.

  This operation may go into an infinite loop if there is a circularity in the parent hierarchy, in which case it is undefined. In Alloy, constraints are logical formulae that are true or false. There is no undefined status. `allParents` is defined as a relation using the transitive closure operator. The `allParents` operation is used in the following OCL constraint.

OCL	`GeneralizableElement` `not self.allParents->includes(self)`
Alloy	`all e: GeneralizableElement |` `e !in e.allParents`
English	Circular inheritance is not allowed.

The OCL constraint above intends to rule out circularity in the parent hierarchy. However, if there is circularity, then `allParents` goes into an infinite loop and is undefined, causing the constraint to be undefined, not false as intended. On the other hand, the Alloy constraint is well-defined and rules out the circularity in the parent hierarchy.

Consider now the following OCL operation.

OCL	`AssociationClass` `allConnections : Set(AssociationEnd);` `allConnections = self.connection->union(self.parent` `->select (s	s.oclIsKindOf(Association))` `->collect (a: Association | a.allConnections)) -> asSet`
Alloy	`all a: AssociationClass |` `a.allConnections = a.*parent.connection`
English	The operation `allConnections` results in the set of all AssociationEnds of the AssociationClass, including all connections defined by its parents (transitive closure).

In this OCL definition, `self.parent` returns a set of AssociationClass objects. This follows from a different constraint (Generalization [1] from UML documentation [2]), which says that an element may be the parent of only another element of the same oclType. Now consider the collection of `a`'s in `(a: Association | a.allConnections)` from the OCL example above. These `a`'s are of oclType AssociationClass, raising the following question. Which definition of `allConnections` should be used? The one for AssociationClass or Association? The Alloy constraint directly expresses the English description, without encountering this difficulty.

- The following example illustrates how the OCL type system might be made simpler.

OCL	`Class`
	`self.allContents->forAll (c |`
	`  c.oclIsKindOf(Class) or`
	`  c.oclIsKindOf(Association) or`
	`  c.oclIsKindOf(Generalization) or`
	`  c.oclIsKindOf(UseCase) or`
	`  c.oclIsKindOf(Constraint) or`
	`  c.oclIsKindOf(Dependency) or`
	`  c.oclIsKindOf(Collaboration) or`
	`  c.oclIsKindOf(DataType) or`
	`  c.oclIsKindOf(Interface))`

Alloy	`all c: Class |`
	`  c.allContents in (Class + Association + Generalization`
	`  + UseCase + Constraint + Dependency + Collaboration`
	`  + DataType + Interface)`

English	A Class can only contain Classes, Associations, Generalizations, UseCases, Constraints, Dependencies, Collaborations, DataTypes, and Interfaces as a Namespace.

Alloy treats classes as sets, allowing set operators such as union to be applied to them directly. This avoids the use of quantification (`forall`) and coercion (`oclIsKindOf`) as needed in OCL.

- The following example illustrates the non-uniform treatment of sets and scalars in OCL. The authors of the semantics document write the following:

```
GeneralizableElement
parent: Set(GeneralizableElement)
parent = self.generalization.parent
```

Technically, the above operation is not allowed in OCL, since `self.generalization` results in a set, and the `collect` construct is missing. This operation should properly be written as shown below.

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

Alloy	`all e: GeneralizableElement |` `  e.parent = e.generalization.parent`
English	The operation `parent` returns a set containing all direct parents.

However, the authors [2] explicitly state that they omit `collect` whenever practical. Uniform navigation through sets and scalars is built in the basic semantics of Al-

loy[1]. Therefore, it allows for simpler expressions without requiring further simplifying assumptions.

- Finally, the following example illustrates the rest of the issues raised in Section 2.

OCL	`Classifier`
	`self.feature->select(a | a.oclKindOf(Attribute))->forall(a |`
	`  not self.allOppositeAssociationEnds->union (self.allContents)`
	`  ->collect(q | q.name)->includes(a.name))`

Alloy	`all c: Classifier |`
	`  all a: c.feature & Attribute |`
	`  a.name !in (c.allOppositeAssociationEnds + c.allContents).name`

English	The name of an Attribute may not be the same as the name of an opposite AssociationEnd or a ModelElement contained in the Classifier.

The Alloy expression can be read as follows: For all classifiers `c`, and all features `a` of `c` that are also Attributes, the name of `a` is not in the set of names of `c`'s opposite AssociationEnds and `c`'s contents.

  - The class `Attribute` cannot be treated directly as a collection in OCL. This makes the coercion operator `oclIsKindOf` necessary, and prevents the use of set operators on classes. On the other hand, Alloy treats `Attributes` as a set and uses set intersection on it directly (`c.feature & Attribute`), resulting in a shorter expression.
  - The quantifiers and set operators are stacked in OCL, and are often textually placed in a way that is not natural. In the example above, the quantifier `forall` is placed after the collection over which it is quantifying. On the other hand, quantifiers in Alloy are placed in the same way as in the English description.
  - Logical operators are not stacked in OCL, and this causes the separation of certain logical and set operators. For example the `not` above begins an expression, but negates the `includes` at its far end. On the other hand, Alloy allows logical operators to be combined with set operators as in `!in`, which means "is not included".

References

[1] Jos Warmer and Anneke Kleppe. "The Object Constraint language. Precise Modelling with UML", Object Technology Series, Addison-Wesley, 1999, p. 8.

[2] "UML Semantics". UML Documentation 1.3, Part 2.
`http://www.rational.com/uml/resources/documentation/index.jtmpl`

[1] In the code, the last reference to parent above is renamed to parent1, because relation names must be unique. This limitation is remedied in the next version of Alloy.

[3] Daniel Jackson. "Alloy: A Lightweight Object Modelling Notation". Submitted for Publication, October 1999.
http://sdg.lcs.mit.edu/~dnj/publications.html

[4] Daniel Jackson. "A Comparison of Object Modelling Notations: Alloy, UML, and Z". August 1999.
http://sdg.lcs.mit.edu/~dnj/publications.html

[5] "Alcoa: Alloy's Constraint Analyzer".
http://sdg.lcs.mit.edu/alcoa

A UML Metamodel in Alloy: Classes

In this appendix, we present the declaration of classes for the UML metamodel in Alloy, as well as some helper relations used in the well-formedness constraints.

```
model uml-core{
  domain {Element, Name,
    VisibilityKind, AssociationClassDom, IndexDom  }
  state{
    partition public, protected, private: VisibilityKind!

    ModelElement: Element+
    //ModelElement attributes
    name: ModelElement! -> Name!

    //associationClass ElementOwnership
    ElementOwnership: AssociationClassDom
    elementOwnership_private: ElementOwnership
    elementOwnership_protected: ElementOwnership //no need for public
    elementOwnership: ModelElement -> ElementOwnership!

    Relationship: ModelElement+
    Association: Relationship+

    AssociationEnd: ModelElement+
    //AssociationEnd attributes
    aggregation: AssociationEnd
    composition: AssociationEnd  //no need for none
    isNavigable: AssociationEnd
    multiplicity_max1: AssociationEnd

    Namespace: ModelElement+
    Feature: ModelElement+
    //Attributes of Feature
    visibility2: Feature -> VisibilityKind

    StructuralFeature: Feature+
    Attribute: StructuralFeature+
```

```
BehavioralFeature: Feature+
//BehavioralFeature attributes
isQuery: BehavioralFeature
//BehavioralFeature operation
hasSameSignature: BehavioralFeature -> BehavioralFeature

Operation: BehavioralFeature+
Method: BehavioralFeature+

GeneralizableElement: ModelElement+
// GeneralizableElement attributes
isRoot: GeneralizableElement
isLeaf: GeneralizableElement
isAbstract: GeneralizableElement
// GeneralizableElement operations
parent: GeneralizableElement -> GeneralizableElement
allParents: GeneralizableElement -> GeneralizableElement

Parameter: ModelElement+
Index: IndexDom+
//Index relation
same: Index -> Index
Constraint: ModelElement+
UseCase: ModelElement+
Collaboration: ModelElement+
Dependency: Relationship+


Classifier: GeneralizableElement+
//Classifier operations
parentClassifier: Classifier -> Classifier
allFeatures: Classifier -> Feature
allOperations: Classifier -> Operation
allMethods: Classifier -> Method
allContents: Classifier -> ModelElement
associations: Classifier -> Association
oppositeAssociationEnds: Classifier -> AssociationEnd
allOppositeAssociationEnds: Classifier -> AssociationEnd
allAttributes: Classifier -> Attribute
specification: Classifier -> Classifier

Class: Classifier+
//Class attribute

AssociationClass: ModelElement+
//AssociationClass operations
parentAssociation: AssociationClass -> Association
```

```
  allConnections: AssociationClass -> AssociationEnd

  Generalization: Relationship+
  //Generalization Attribute
  discriminator: Generalization -> Name

  Interface: Classifier+
  DataType: Classifier+

  Component: Classifier+
  //Component operations
  parentComponent: Component -> Component
  allResidentElements: Component -> ModelElement

  //Association class between Component and ModelElement
  ElementResidence: AssociationClassDom
  component: ElementResidence -> Component?
  modelElement(~elementResidence): ElementResidence -> ModelElement?
  visibility1: ElementResidence -> VisibilityKind

  //Stereotypes
  Type: Class+


  // Relationships
  connection(~association): Association! -> AssociationEnd+
  namespace(~ownedElement): ModelElement -> Namespace?
  type(~associationEnd): AssociationEnd -> Classifier!
  ptype: Parameter -> Classifier!
  feature(~owner): Classifier! -> Feature
  specialization(~parent1): GeneralizableElement! -> Generalization
  generalization(~child): GeneralizableElement! -> Generalization
  index: BehavioralFeature? -> Index
  parameter: Index? -> Parameter?
  //specification: Method -> Operation!
  powertypeRange: Classifier? -> Generalization
  constrainedElement: Constraint -> ModelElement+
  resident: Component -> ModelElement
}


// OPERATION DEFINITIONS
cond GeneralizableElementOp {
  //Definition of parent
  all e: GeneralizableElement |
        e.parent = e.generalization.parent1

  //Definition of allParents
```

```
  all e: GeneralizableElement |
        e.allParents = e.+parent

}

 cond AssociationClassOp{
   //include operations
   GeneralizableElementOp

   //Definition of allConnections
   all a: AssociationClass | a.parentAssociation = a.parent & Association

   all a: AssociationClass |
            a.allConnections = a.+parentAssociation.connection
 }

 cond BehavioralFeatureOp{
   all b1, b2: BehavioralFeature | b1.index != b2.index

   all b: BehavioralFeature | all i: b.index |
           some i.parameter

   //Definition of same - equivalence relation
   all i: Index | i in i.same
   all i1, i2: Index | i1 in i2.same -> i2 in i1.same
   all i1, i2, i3: Index | i1 in i2.same && i2 in i3.same ->
          i1 in i3.same

   //Definition of hasSameSignature
   all b: BehavioralFeature |
      b.hasSameSignature =  {b1: BehavioralFeature |
         b.name = b1.name &&
        b1.index in b.index.same &&
         b.index in b1.index.same &&
        (all i: b.index | all i1: b1.index |
              i1 in i.same -> i.parameter.ptype = i1.parameter.ptype)}

 }

 cond ClassifierOp{
   //include operations
   GeneralizableElementOp

   all c: Classifier | c.parentClassifier = c.parent & Classifier

   //Definition of allFeatures
   all c: Classifier | c.allFeatures = c.+parentClassifier.feature
```

```
  //Definition of allOperations
  all c: Classifier | c.allOperations = c.allFeatures & Operation

  //Definition of allMethods
  all c: Classifier | c.allMethods = c.allFeatures & Method

  //Definition of allContents
  all c: Classifier |
    c.allContents = c.+parentClassifier.ownedElement &
         {e: ModelElement | e.elementOwnership !in elementOwnership_private}

  //Definition of associations
  all c: Classifier | c.associations = c.associationEnd.association

  //Definition of oppositeAssociationEnds
  all c: Classifier |
    c.oppositeAssociationEnds =
     {e: AssociationEnd | e in
       {a: c.associations | one c1 : a.connection.type | c1 = c}.connection &&
         c !in e.type} +
       {e: AssociationEnd | e in
         {a: c.associations | some c1,c2: a.connection.type |
           c1 = c && c2 = c && c1 != c2}}

  //Definition of allOppositeAssociationEnds
  all c: Classifier |
     c.allOppositeAssociationEnds = c.+parent.oppositeAssociationEnds

  //Definition of allAttributes
  all c: Classifier |
    c.allAttributes = c.allFeatures & Attribute

}

 cond ComponentOp{
  all c: Component | c.parentComponent = c.parent & Component

  //Definition of allResidentElements
  all c: Component |
     c.allResidentElements = c.*parentComponent.resident &
           {e: ModelElement | e.elementResidence.visibility1 !in private}
}
```

B UML Metamodel in Alloy: Constraints

In this appendix, we present the well-formedness constraints for the UML metamodel in Alloy.

```
// WELL-FORMEDNESS CONDITIONS
inv AssociationWF{
  // connection multiplicity
  all a: Association | not one a.connection

  // Association [1]
  all a: Association | all e1,e2: a.connection |
          e1.name = e2.name -> e1 = e2

  // Association [2]
  all a: Association | sole a.connection & (aggregation + composition)

  // Association [3]
  all a: Association | some e1,e2,e3: a.connection |
   (e1 != e2 && e1 != e3 && e2 != e3) ->
    a.connection !in (aggregation + composition)

  // Association [4]
  all a: Association | a.connection.type in a.namespace.ownedElement

}

inv AssociationClassWF{
  //Multiple inheritance
  AssociationClass = Association & Class

  //Include operations
  AssociationClassOp
  ClassifierOp

  //AssocitationClass [1]
  all a: AssociationClass |
    no (a.allConnections.name & (a.allFeatures & StructuralFeature).name)

  //AssocoationClass [2]
  all a: AssociationClass | a !in a.allConnections.type
}

inv AssociationEndWF{
  //agregation and composition are disjoint
  no aggregation & composition
```

```
  //AssociationEnd [1]
  all e: AssociationEnd |
          (e.type in (Interface + DataType)) ->
          (e.association.connection - e) !in isNavigable

  //AssociationEnd [2]
  all e: AssociationEnd | e in composition ->
                          e in multiplicity_max1
}

inv BehavioralFeatureWF{
  //BehavioralFeature [1]
  all b: BehavioralFeature | all p1,p2 : b.index.parameter |
           p1.name = p2.name -> p1 = p2

  //BehavioralFeature [2]
  all b: BehavioralFeature |
          b.index.parameter.ptype in b.owner.namespace.ownedElement
}

inv ClassWF{
  //include operations
  ClassifierOp

  //Class [1]
  all c: Class | c !in isAbstract ->
     all o: c.allOperations | some m: c.allMethods |
         o in m.specification

  //Class [2]
  all c: Class | c.allContents in
         (Class + Association + Generalization + UseCase +
          Constraint + Dependency + Collaboration + DataType + Interface)
  }

inv ClassifierWF{
  //include operations
  ClassifierOp

  //Classifier [1]
  all c: Classifier | all f,g: c.feature |
     (((f in Operation && g in Operation) ||
       (f in Method && g in Method))
      &&
      g in f.hasSameSignature)
     -> f = g
```

```
    //Classifier [2]
    all c: Classifier | all p,q :(c.feature & Attribute) |
       p.name = q.name -> p = q

    //Classifier [3]
    all c: Classifier | all p,q: c.oppositeAssociationEnds |
       p.name = q.name -> p = q

    //Classifier [4]
    all c: Classifier | all a: c.feature & Attribute |
            a.name !in (c.allOppositeAssociationEnds + c.allContents).name

    //Classifier [5]
    all c: Classifier | all o: c.oppositeAssociationEnds |
           o.name !in (c.allAttributes + c.allContents).name

    //Classifier [7]
    all c: Classifier | all g1, g2: c.powertypeRange |
             g1.discriminator = g2.discriminator

  }

inv ComponentWF{
   //include operations
   ComponentOp

   //Component [1]
   Component.allContents in Component

   //Component [2]
   all e: Component.allResidentElements | e in (DataType + Interface +
             Class + Association + Dependency + Constraint)

  }

inv ConstraintWF{
  //Constraint [1]
  all c: Constraint | c !in c.constrainedElement
}

inv DatatypeWF{
  //DataType [1]
  all d: DataType | all f: d.allFeatures | f in Operation & isQuery

  //DataType [2]
  all d: DataType | no d.allContents
}
```

```
inv ElementOwnershipWF{ //visibility constraint
  no elementOwnership_private & elementOwnership_protected
}

inv GeneralizableElementWF{
   //include operations
   GeneralizableElementOp


  //GeneralizableElement [1]
  all e: GeneralizableElement | e in isRoot -> no e.generalization

  //GeneralizableElement [2]
  all e: GeneralizableElement | no (e.parent  & isLeaf)

  //GenrelizableElement [3]
  all e: GeneralizableElement | e !in e.allParents

  //GeneralizableElement [4]
  all e: GeneralizableElement | all g: e.generalization |
            g.parent1 in e.namespace.allContents
 }

inv GeneralizationWF{
  //Generalization [1]
  all g: Generalization |
    g.parent1 in Class && g.child in Class ||
    g.parent1 in Classifier && g.child in Classifier ||
    g.parent1 in AssociationClass && g.child in AssociationClass ||
    g.parent1 in Association && g.child in Association
}

inv InterfaceWF{
  //Interface [1]
  all i: Interface | i.allFeatures in Operation

  //Interface [2]
  no Interface.allContents

  //Interface [3]
  Interface.allFeatures.visibility2 in public
}

inv MethodWF{
  //Method [1]
  all m: Method | m.specification in isQuery -> m in isQuery
```

```
  //Method [2]
  all m: Method | m.specification in m.hasSameSignature

  //Method [3]
  all m: Method | m.visibility2 = m.specification.visibility2

  //Method [4]
  all m: Method | m.specification in m.owner.allOperations

  //Method [5]
  all m: Method | (m.owner.allOperations &
                     {o: Operation | o in m.hasSameSignature})
                   in m.specification.owner.allOperations
}

inv NamespaceWF{
  //Namespace [1]
  all n: Namespace | all e1,e2: n.allContents |
   (e1 !in Association  && e2 !in Association &&  one e1.name && one e2.name &&
               e1.name = e2.name) -> e1 = e2

  //Namespace [2]
  all n: Namespace | all a1, a2: n.allContents & Association |
   (a1.name = a2.name && a1.connection.type = a2.connection.type) -> a1 = a2
}

inv StructuralFeatureWF{
  //StructuralFeature [1]
  all s: StructuralFeature | s.type in s.owner.namespace.allContents

  //StructuralFeature [2]
  all s: StructuralFeature | s.type in (Class + DataType + Interface)

}

inv TypeWF{
  //Type [1]
  no Type.feature & Method

  //Type [2]
  all t:Type | t.parent in Type
}
}
```