DOI: 10.2298/CSIS110210045V

Using Lightweight Formal Methods to Model Class and Object Diagrams

Fernando Valles-Barajas

Department of Information Technology, Faculty of Engineering, Universidad Regiomontana
15 de Mayo 567 Pte., C.P. 64000 Colonia Centro, Monterrey, Nuevo León, México
phone: +52 81 8220-4733
fernando.valles@acm.org, fernando.valles@ieee.org

Abstract. In this paper a formal model for class and object diagrams is presented. To make the model the author used Alloy, which is a three-in-one package: a modeling language that constructs software models, a formal method that guides the construction of software models and an analyzer that helps find inconsistencies in software models. In the proposed model the entities that form class and object diagrams, as well as the rules that govern how these elements can be connected, are specified.

Keywords: Alloy, Formal Methods, UML.

1. Introduction

A model is a simplified representation of a system; it is useful for documenting, modeling, communicating and analyzing software systems [31]. One of the mechanisms available for making a software model is Unified Modeling Language (UML), which uses a graphical notation [30]. There are several diagrams in UML, each of these have a specific purpose; for example a class diagram represents the entities of a system as well as the relations between them. A class diagram also represents the attributes of system entities, operations carried out by these entities and the responsibilities assigned to them. There are several integrated development environments (IDEs) that facilitate the process of making UML diagrams; for instance ArgoUML, Borland Together and IBM Rational Rose.

Motivation of the paper: In some of the IDEs mentioned above, a novice user can build an inconsistent UML diagram; for example it is possible to build a class diagram with two classes associated by composition in both directions with Borland Together; in other words the *whole-part* class is also a component of the *part-of* class. In the personal opinion of the author of this paper, the reason for this problem is a weak design of the IDEs mentioned above. Because formal methods have been used to construct models without ambiguity and inconsistencies and to have a strong design [28], in this paper the formal specification for a tool that builds class and object diagrams is proposed. Formal methods use mathematical notations to specify and analyze software models

[17]. The formal method used in this paper is Alloy; which was developed by the Massachusetts Institute of Technology (MIT) software design group. As will be seen, this formal method is also a modeling language that creates software models in a step-by-step style as well as an analyzer that helps to find inconsistencies in software models.

Related works: UML is a modeling language used in software processes (e.g. Rational Unified Process (RUP)) to generate software models. Because of its simplicity, this language has been widely applied in the software industry. One of the disadvantages of UML models is that these artifacts can be interpreted differently by two members of a software team.

Formal methods are useful techniques to formally specify the syntax of UML diagrams. Models obtained using these techniques are unambiguous. For example, in [14] Z language is used to specify the syntax of class diagrams. The author remarks that Z is especially useful to formally specify class diagrams involving recursive structures.

Object-Z is an extension of Z language that includes object-oriented concepts (inheritance, polymorphism and encapsulation among others). This language is used in [20] to specify the syntax of class diagrams. Because UML and Object-Z are based both on object-oriented concepts, the mapping between these two languages is more natural than the mapping between UML and Z. The definition of UML classes in Z and Object-Z language supports this idea. UML classes specify the structure, behavior and associations shared by a set of objects. While in [14], UML classes were modeled using two Z schemes; one for defining class structures and other for defining behavior, in [20] using Object-Z only one container was necessary to define UML classes. Besides of this, an Object-Z element defining classes can inherit variables and operations from one element that has already been defined. Another paper that formalizes UML class diagrams using Object-Z is [21].

In [2] the authors compare several approaches based on Z and Object-Z to model class diagrams and one of the conclusions made is that thanks to the object-oriented concepts of Object-Z, this modeling language produces more concise models than Z language.

Another modeling language that has been used to specify the syntax of UML class diagrams is the Prototype Verification System Specification Language (PVS-SL), see [4]. In [24] the authors present a tool that analyzes the syntax and semantics of Object Constraint Language (OCL) constraints together with UML models and translates them into PVS-SL. The authors tested the proposed tool representing the Sieve of Eratosthenes algorithm, which is an algorithm that finds prime numbers, in a UML class diagram and OCL constraints and then this model was converted to a PVS-SL model.

In the analysis phase, it is mandatory to have a notation that allows software developers to generate requirement models. These models should be done so that good communication between users and the development team is established. As the authors of [15] state, UML has demonstrated to be a good option in developing requirements models, however these models cannot be analyzed and reasoning based on these models is difficult. To overcome these problems the authors of [15] propose a tool that receives Extensible Markup Language (XML) class diagram specifications as input and generates RAISE Specification Language (RLS) class diagram specification as output. The tool is able to analyze and reason of the original class diagrams. It is important to mention that modeling tools (e.g. ArgoUML) can store class diagrams in an XML format; so the input for the proposed tools is easy to obtain.

OCL is the de facto modeling language for adding constraints to UML diagrams (see [1]). This language was developed because UML constructs alone cannot precisely model situations occurring in real world problems [37].
USE is a UML-based Specification Environment that allows software developers to model UML class diagrams along with OCL constraints ([16]). USE makes it possible to check the consistency of UML models by generating instances of class diagrams.

The authors in [8] argue that OCL has weak semantics and that reasoning from UML diagrams along with OCL constraints is difficult. These authors state that by transforming from OCL to the Isabelle/HOL theorem prover, property analysis of UML diagrams is possible.

Models consist of a dynamic and a static part. UML has a variety of diagrams covering these two parts. For example the dynamic part may be modeled by state machine diagrams and the static part by class diagrams. In [26] the authors present a technique to transform class and state machine diagrams into B modeling language. Models represented in the B notation can be analyzed.

Papers [32], [33], [35], [34], [36] demonstrate that Alloy, which is the modeling language used in this paper, has been successfully used to formally specify UML diagrams. In [32] the author uses Alloy to model use case diagrams. In [33] state machine diagrams are formalized and an extended version of this paper was later presented in [35]. In [34] a model of System Modeling Language (SysML) requirements diagrams using Alloy is included. It is important to mention that SysML is a UML profile.
UML collaborations model interactions between entities working together to accomplish a specific task. In [36], the author formally specify Collaborations.

Other techniques that do not apply formal methods have been used to verify the correctness of UML diagrams. In [19] the authors use constraint programming for this purpose. UML diagrams along with OCL constraints are first specified as a constraint satisfaction problem and then properties of UML class diagrams are verified with the UMLtoCSP tool.

Methods having roots in artificial intelligence have also been used to specify UML diagrams and to later reason from these models. For example description logic, which is used in artificial intelligence to represent knowledge, is used in [6] and [10] with this objective.

Coalgebra is used in computer science to specify the states of a system. This technique is applied in [27] to represent the semantics of class and use case diagrams. Associations among classes are interpreted as coalgebraic observers. The generalization hierarchy of classes is specified by the inheritance morphism among them.

Graph theory is applied in [22] to model class diagrams as graphs. Class diagrams are mapped as nodes with connecting edges.

Formal methods are based on mathematical logic; for example, Alloy is a modeling language based on set theory, first order logic and relational logic. Equational logic was developed to help in the formal development of programs. A novel method based on equational logic to model class diagrams is presented in [11].

Paper [9] does not apply formal methods to UML class diagrams but it formalizes aggregation and composition relationships. As the authors mention, there are some UML concepts that are not well specified in the Object Management Group (OMG) documentation of UML. Two examples of these poorly specified concepts are the aggregation and composition relationships. The paper models these two relationships as subclasses of a more generic class called the whole-part relationship. In UML 1.x these relationships were modeled as meta-attributes. With the addition in the UML meta-model of one class for each relationship, related characteristics of the aggregation and composition relationships can be attached to theses classes. The following characteristics of these two new classes are considered in the author's proposal: shareability, separability, mutability, configurationality, lifetime dependency and existential dependency.

As the reader may notice, the papers reported so far are based on the transformation of UML class diagrams to a more formal notation to get analyzable and precise models. According with [12], the disadvantage of this approach is that this transformation requires a strong background in mathematics and, unfortunately, most practitioners do not fulfill this requirement. In [12], the author proposes a technique based on the manipulation of UML class diagrams by using some transformation rules. These rules allow to know if one class diagram is a conjecture of another class diagram. The method presented by the author does not require a strong knowledge of mathematics.

In [23] the authors study the application of graph transformation, which a mature field, to describe the semantics of class, object and state diagrams. The state of a system is represented by object diagrams and these are in turn represented by graphs. A change in the system state is described by using two graphs (representing previous and later states) and by transformation rules. The paper does not present analysis or reasoning of class diagrams based on graph transformation.

Structure of the paper: In this section the motivation of the paper has been given. The following section contains some basic concepts of Alloy; which is the formal method that will be used to model class and object diagrams. Section 3 introduces the necessary concepts of class and object diagrams to understand the model proposed in the paper. Section 4 contains the formal model. In section 5 a comparison of Alloy with other formal methods is presented. Concluding remarks are given in section 6.

2. A brief introduction to Alloy

Alloy is a modeling language that has three mathematical tools for making software models: first order logic, relational calculus and set theory [18]. A system is modeled by representing each system entity and the relations between them. The entities are represented as signatures, which are similar to classes, and the relations as fields of the signatures. Constraints between entities can be specified near signatures as signature facts or outside signatures as facts. Functions are used to specify reusing expressions. Basic constraints can be specified in the relations between signatures by using multiplicity keywords. Alloy has the following multiplicity keywords: **one** (exactly one), **lone** (zero or one), **some** (one or more) and **set** (any number). Properties of the model elements can be expressed by using predicates. Alloy has several relational calculus operators that allow to construct complex expressions; for instance the $\wedge$ and $*$ respectively denote the transitive closure and the reflexive-transitive closure of a relation and they are useful for constructing expressions in a relational calculus style. This modeling language also has operators to represent expressions using first order logic; e.g. implication ($\Rightarrow$), not (!), and ($\&\&$), or ($||$) and the bi-implication ($\Leftrightarrow$). The operators for manipulating sets are: in, union ($+$), intersection ($\&$), difference ($-$) and equality ($=$). By using all these operators a modeler can specify software models without ambiguity.

A model in Alloy captures not only the static view of the system but also its dynamic view. It is possible to specify operations that affect the state of the system by using predicates. When a predicate is used to model an operation, the pre-conditions and the post-conditions must be specified.

An attractive characteristic of Alloy is that it does not need a tool to type the formal specification; as will be seen this is written using American Standard Code for Information Interchange (ASCII) characters.

3. Class diagrams

A class diagram is one of the static diagrams of UML [5]. It is used to represent the entities of a system and the relations between them. A more detailed class diagram can include the features of the entities as well as their responsibilities. There are two types of features: structural and dynamic [13]. Structural features can be subdivided in attributes and associations. Attributes correspond to variables in programming languages. Due to the fact that the associations between classes are represented as variables in programming languages, these are also considered to be structural features. The dynamic part of the classes are the operations, which are implemented by methods in a programming language. There are five types of relations between classes: association, aggregation, composition, generalization and dependency.

An association represents a relation between objects of the same level [29]; this is represented using a solid line connecting two related classes.

Aggregation and composition are types of associations [31]. They are useful for representing an entity that is composed of smaller entities. These relations can be logical or physical. The classes that participate in these kinds of associations belong to different abstraction levels; one is the *whole-part* and the other is the *part-of*. Instead of using an association labeled with *whole-part* and *part-of* strings, the solid line that joins two related classes is adorned with a diamond near the *whole-part*. Three important characteristics of the composition relation are:

- if the *whole-part* is destroyed the smaller parts are also destroyed.
- a *part-of* may be a part of only one composite at any time [7].
- the diamond near the *whole-part* is filled (in an aggregation relation this diamond is empty).

Generalization is the process of finding the common features of some classes and then putting these common features into a class. This process is useful to inherit common features from one class to others. The class that passes features is called superclass and the classes that receive the features are called subclasses.

Generalization is also a binary relation between a superclass and a subclass. The generalization relationship is represented as a solid line with a hollow arrowhead at the superclass end. In a class diagram several generalizations are usually arranged forming an inheritance hierarchy; this should not be either too deep or too wide [7]. The set of classes above a class c in the inheritance hierarchy are ancestors of this class and they are obtained by using the transitive closure operator. The set of classes below a class c in the inheritance hierarchy are descendants of this class and they also are obtained by using the transitive closure operator [31]. When an instance is generated from a class c, no matter how many ancestors are in this class, only one object is generated with the features of class c and the ancestors of this class [3]. A class inherits the features of its ancestors as well as their relations [25]. Classes in an inheritance hierarchy can be divided into two types: abstract and concrete. It is not possible to generate an instance from an abstract class. Because of this when a superclass is abstract then the subclasses of this class form a partition [25].

The dependency relation represents the situation in which a change in one element affects other elements. An example of the dependency relation is when a class sends a message to other class; if the receiver of the message changes its interface, then the sender must be changed so that the two classes are still able to communicate. Other examples of dependencies are when a class c_1 has a class c_2 as a parameter or as a local variable for one of its operations or when a class c_2 is in the global scope of a class c_1. According with [13] the dependency relation is anti-transitive. This is formally defined as: $\forall c_1, c_2, c_3 : C\ (c_1 R c_2\ \&\ c_2 R c_3) \Rightarrow not\ (c_1 R c_3)$.

4. A formal model for object and class diagrams

In this section a formal model for class and object diagrams is given. The model was divided in five parts: in the first part an initial model is presented, the second part specifies the association and dependency relations, the third part presents the generalization relation. The four part specifies the aggregation and composition relation. The last part formalizes the operation definition.

Convention used in this paper: In the models presented in this section, for the purpose of clarity, the keywords of the modeling language are printed in **bold font**.

4.1. The initial model

Fig. 1 presents an initial model for class and object diagrams. Line 1 specifies the name of the model (classModel) and the directory for this model (umlMetamodels). Next, in lines 2 and 3, some Alloy libraries are included in the model. Alloy has some built-in libraries that can be used to save time; in this case the model is using two libraries; one to manipulate booleans and another to manipulate relations. Line 6 defines a signature for class diagrams. As was mentioned in section 2 the relations between entities (represented in the model as signatures) are specified using fields of signatures; for instance, the *classes* field of the signature *ClassDiagram* denotes a binary relation between the *ClassDiagram* signature and the *Class* signature. Multiplicity keywords can be used in a relation to constrain the number of participants; for example, the multiplicity keyword **some** in the *classes* field means that a class diagram has one or more classes. Multiplicity keywords can be used also to constraint the instances that can be generated from one signature; the keyword **one** in the *ClassDiagram* signature indicates that only one instance from this signature can be generated.

Some ternary relations are defined in the *ClassDiagram* signature; for example the *association* field is a ternary relation of the form:
association: ClassDiagram→classes→classes. With the definition of this relation, a class diagram knows all classes that are related by the association relation. A model can be documented in Alloy using comments; for example in line 12 a comment was defined (a comment in Alloy begins with a double-dash). The last line of fig. 1 is the **run** command; by using this command the modeler can generate instances of the model to see if there are inconsistencies.

It is important to mention that in the initial model, the inheritance relation was not defined inside the object diagram signature; the reason for this is to remark the idea that the inheritance relation is really a relation between classes (see section 3).

4.2. Association and dependency relations

Fig. 2 contains the part of the model where association and dependency relations are specified. In line 1 one function, which is a reusing expression, is defined. The function *validRelation* checks if two objects o_1 and o_2 can be related using some specific relation. In line 5 the predicate *antiTransitive* is defined to constrain the dependency relation. As was explained in the previous section, Alloy has a module to manipulate relations, but the anti-transitive property is

```
module umlMetamodels/classModel
open util/boolean as booleans
open util/relation as relations

sig Name{}

sig Class{ name: Name }

one sig ClassDiagram{
  classes: some Class,
  association, dependency, inheritance, aggregation,
    composition: classes->classes
  }

sig Object{
  -- models anonymous objects
  name: lone Name,

  type: Class
  }

one sig ObjectDiagram{
  objects: some Object,
  association, dependency, aggregation,
    composition: objects->objects
  }

showInstance: run{} for 7
```

Fig. 1. Initial model

not defined in that module. In signatures *ClassDiagram* and *ObjectDiagram* the dependency relation was constrained to be anti-transitive.

4.3. Generalization relation

Figs. 3, 4 and 5 present the part of the model where the generalization relation is specified. In line 1 of fig. 3 the signature *GeneralizationStructure* was defined; this signature represents an inheritance hierarchy. The field *setName* is used to document the criterion that was used to form a classification; in early versions of UML this concept was called discriminator. Some constraints over the generalization structure were defined in line 6. The *isComplete* field is used to indicate that all subclasses of class c have been defined. The *isDisjoint* field implies that inheritance of a class c from two subclasses of a disjoint generalization structure is not allowed. The *isOverlapping* field is the opposite of the *isDisjoint* field. Two signature facts were defined in the signature *GeneralizationStructure*. Line 9 specifies that superclasses and subclasses defined in the *GeneralizationStructure* signature must belong to the *ClassDiagram* signature. Line 10 declares that there must be a mapping, in the *ClassDiagram* signature, between the superclasses and the subclasses defined in the *GeneralizationStructure* signature.

```
fun validRelation[ r: univ->univ ]:Object->Object{
  { o1, o2: Object |
    o1.type->o2.type in r.Class->Class.r }
  }

pred antiTransitive[r: univ->univ, S: set univ]{
  all s1, s2, s3: S |
    (s1->s2 + s2->s3) in r => s1->s3 not in r
  }

one sig ClassDiagram{

  -- as before }
  {
  antiTransitive[dependency, classes]
  }

one sig ObjectDiagram{

  -- as before }
  {
  association in
    validRelation[ClassDiagram.association]

  dependency in
    validRelation[ClassDiagram.dependency]

  antiTransitive[dependency, objects]
  }
```

Fig. 2. The association and dependency relations

As can be seen in fig. 3, a class is composed by structural and dynamic features. The static features defined in *Class* signatures were properties; associations were defined in the *ClassDiagram* signature as relations (see fig. 1). The *isSubstitutable* boolean field is used to indicate if a subclass is used to substitute one of its superclasses. A class that does not stand alone and is used to add behavior and structure to other classes is called a mixin class; it is used in multiple inheritance relationship [31].

Fig. 4 presents part II of the generalization relation. The properties for the generalization relation are defined in the *ClassDiagram* signature: acyclic, irreflexive and antisymmetric. Two constraints about the form of the inheritance hierarchy were defined: an inheritance hierarchy should not be too deep or too wide. Line 14 constrains the model to only single inheritance. This line was written as a comment because most programming languages do accept multiple inheritance. Line 19 of fig. 4 defines the *isDisjoint* property defined in fig. 3. In line 26 a signature fact was defined to specify that a class not only inherits the features of its ancestors but also their associations. In fig. 4 a predicate was defined to model the *isComplete* property of the generalization relation; in this predicate the precondition and post condition were defined.

The last part of the generalization relation is shown in fig. 5. This figure defines the concept of classification. An object usually belongs to a class but

```
1  sig GeneralizationStructure{
2    superClass: Class,
3    subClasses: some Class,

4    -- set names are also known as discriminators
5    setName: Name,
6    isIncomplete, isDisjoint, isOverlapping: Bool}
7    {
8    let cd = ClassDiagram{
9      (superClass + subClasses) in cd.classes

10     some c: (cd.inheritance).(cd.classes) |
11       (c = superClass) &&
12       (subClasses = cd.inheritance[c])
13     }
14   }

15 abstract sig Feature{}
16 sig Property, Operation extends Feature{}

17 sig Class{
18   -- as before

19   features: some Feature,

20   isSubstitutable,
21   isAMixin,

22   isRoot, isLeaf, isAbstract: Bool}
23   {
24   let cdi = ClassDiagram.inheritance{
25     --  a root class that has ancestors
26     -- is not permitted
27     no cdi.this => isRoot = True
28       else isRoot = False

29     --  a leaf class that has descendants
30     -- is not allowed
31     no this.cdi => isLeaf = True
32       else isLeaf = False

33     -- an abstract class must have descendants
34     -- and it can not be a leaf class
35     some (this & cdi.this) && (isLeaf != True)
36       => isAbstract = True
37     else
38       isAbstract = False
39     }

40   isAMixin = True => isAbstract = True
41   }
```

Fig. 3. The generalization relation; part I

there is no restriction indicating that an object cannot belong to more than one class. The field *classification* stores the possible classes to which an object can belong. Line 16 models the fact that an object contains the features of its type (class) and ancestors of this type.

```
one sig ClassDiagram{
  -- as before }
  {
  -- as before

  -- constraints on the generalization relation --

  acyclic[inheritance, classes]
  irreflexive[inheritance]
  antisymmetric[inheritance]

  -- controls the width of the inheritance hierarchy
  all c: inheritance.classes | #(c.inheritance) =< 6

  -- controls the depth of inheritance hierarchy
  all c: classes | #(c.^inheritance) =< 5

  -- avoids multiple inheritance
  -- all c: classes.inheritance | one inheritance.c

  all c: inheritance.classes |
    some ge: GeneralizationStructure |
      c = ge.superClass

  -- models the disjoint property of generalization
  let i = inheritance |
    all c1: i.classes | #(c1.i) > 1 &&
      (superClass.c1).isDisjoint in True =>
        no c2: classes.i |
          #(i.c2) > 1 && i.c2 in c1.i

  --  a class inheritances the associations of
  -- its ancestors
  all c: classes | all a: getAncestors[c] |
    c->a.association in association
  }

-- models the isComplete property of generalization
pred addDescendant[cd, cd': ClassDiagram, f, s: Class]{
  -- pre conditions
  cd != cd'
  f in cd.classes
  some ge: GeneralizationStructure |
    ge.superClass = f && ge.isIncomplete = True
  f->s not in cd.inheritance

  -- post conditions
  cd'.inheritance = cd.inheritance + f->s
  }

fun getMultipleInheritance[ cd: ClassDiagram ]:
  set Class{
  { c: (cd.classes).(cd.inheritance) |
    #(cd.inheritance).c > 1  }
  }
```

Fig. 4. The generalization relation; part II

4.4. Aggregation and composition relation

The last relations that were defined were the aggregation and composition relations. Both relations have the following properties: acyclic, irreflexive and an-

```
sig Object{
  -- as before

  features: some Feature,

  classification: set Class,
  multipleClassification: Bool}
  {
  type in ClassDiagram.classes

  -- models multiple classification
  classification in
    getGeneralizationStructure[ type ].subClasses
  #classification > 1 =>
    multipleClassification in True

  --  an object contains the features of
  -- its type (class) and the ancestors of
  -- this type
  let i = ClassDiagram.inheritance |
    -- * is the reflexive transitive closure operator
    features =
      { f: Feature | f in *i.type.features }
  }

fun getAncestors[c: Class]: set Class{
  -- ^ is the transitive closure operator
  { x: Class | x in ^(ClassDiagram.inheritance).c }
  }

fun getDescendants[c: Class]: set Class{
  { x: Class | x in c.^(ClassDiagram.inheritance) }
  }

fun getGeneralizationStructure[ c: Class ]:
  set GeneralizationStructure{
  { ge: GeneralizationStructure | ge.superClass = c }
  }

pred mutateClassification[ o: Object, c: set Class ]{
  o.classification = c
  }
```

Fig. 5. The generalization relation; part III

tisymmetric. Two operations were defined related to these relations. The *deleteObject* predicate models the fact that an object that is *part-of* another object in a composition relation cannot exist outside the *whole-part*.

4.5. Operations

Fig. 7 contains the definition of operations. This model shows how Alloy deals with sequences. As the reader may notice, an operation is composed of a sequence of parameters (see the Operation signature). The specification of an operation could have been defined as a set of parameters, but this would have been incorrect because the parameter positions within an operation do matter.

```
one sig ClassDiagram{

  -- as before }
  {
  -- as before

  acyclic[aggregation, classes]
  irreflexive[aggregation]
  antisymmetric[aggregation]

  acyclic[composition, classes]
  irreflexive[composition]
  antisymmetric[composition]
  }

one sig ObjectDiagram{

  -- as before }
  {
  -- as before

  composition in
    validRelation[ClassDiagram.composition]
  aggregation in
   validRelation[ClassDiagram.aggregation]

  acyclic[aggregation, objects]
  irreflexive[aggregation]
  antisymmetric[aggregation]

  acyclic[composition, objects]
  irreflexive[composition]
  antisymmetric[composition]
  }

pred delObject[ od, od': ObjectDiagram, o: Object]{
  -- pre conditions
  o in od.objects
  not o in (od.aggregation).(od.objects)

  -- post conditions
  od'.objects = od.objects - o
  }

--   this operation takes into account the
-- composition relation
pred addPart[ od, od': ObjectDiagram, w, p: Object ]{
  -- pre conditions
   (w + p) in od.objects
  w->p in validRelation[ClassDiagram.composition]
  not w->p in od.composition
  -- no sharing
  not p in od.composition[Object]

  -- post conditions
  od'.composition = od.composition + w->p
  }
```

Fig. 6. The aggregation and composition relation

```
open util/boolean as booleans

enum VisibilityKind{ Private, Public, Protected }

sig Name{}

sig Type{}

sig Parameter{
  name: Name,
  type: Type
  }

sig Operation{
  name: Name,
  visibilityKind: VisibilityKind,
  parameters: seq Parameter,
  isAbstract: Bool }
  {
    -- parameter names should be unique
    all disj i, j: #parameters | no (parameters[i].name & parameters[j].name)
  }


sig Class{
  ops: some Operation,
  isAbstract: Bool }
  {
    -- operations should have different signatures
    all disj op1, op2: ops | op1.name = op2.name
      && #(op1.parameters) = #(op2.parameters) => all i: #op1.parameters |
        op1.parameters[i].type = op2.parameters[i].type => op1 = op2

    -- if a class has an abstract operation, it must be abstract
    ops.isAbstract = True => isAbstract = True

    -- abstract classes have at least one abstract operation
    isAbstract = True => some o: ops | o.isAbstract = True
  }

fact{
  -- classes should have different operations
  all disj c1, c2: Class | no (c1.ops & c2.ops)
  }

showInstance: run{} for 4
```

Fig. 7. Definition of operations

5. Comparison of Alloy with other formal methods

A valid question the reader might have is: what are the advantages of Alloy in comparison to other formal methods? To answer this question let us analyze an excerpt of the meta-model, which is presented in fig. 8. In this fig., the inheritance relationship between classes and objects is emphasized.

One of the most important characteristics of Alloy is that it is possible to get an instance of the model without actually writing any single line of code. Fig. 9 shows an instance of the meta-model except (the name of the inheritance relation was changed to *isATypeOf*). The reader may notice that this fig. presents some problems, one of these is that the instance of the object diagram does not correspond to the definition of the class diagram instance; the class diagram instance states that Class3 is a type of Class1 (Class3 is a subclass of Class1); however in the object diagram instance Object2, which is an instance of Class3, inherits from Object3, which an instance of Class2. In order for the object diagram instance to correspond to the class diagram instance, the type of Object3 should be Class1.

To fix this problem, lines 5 and 6 of fig. 10 were added to the model. After these lines were added another instance was generated; see fig. 11. The previous problem was solved but the model still has some inconsistencies, e.g. some classes and objects inherit to themselves (Class1, Class3 and Object3). Lines 8 and 15 of fig. 10 solved this problem. After these lines were added to the model, no flaws were found (see fig. 12).

Even thought the last instance does not present the error that class C_1 inherits from class C_2 and at the same time class C_2 inherits from class C_1, it is possible using the meta-model to generate an instance having this inconsistency. Lines 8 and 15 were added to avoid this possible problem.

```
module umlMetamodels/classModel

sig Name{}

sig Class{ name: Name }

one sig ClassDiagram{
  classes: some Class,
  inheritance : classes->classes
  }

sig Object{
  name: lone Name,
  type: Class
  }

one sig ObjectDiagram{
  objects:  some Object,
  inheritance: objects->objects
  }
```

Fig. 8. An excerpt of the meta-model

6. Conclusions

In this paper a formal specification for a tool that builds class and object diagrams was given. As can be seen in the model of this paper, Alloy uses propositional logic, relational calculus and set theory to build software models. Even

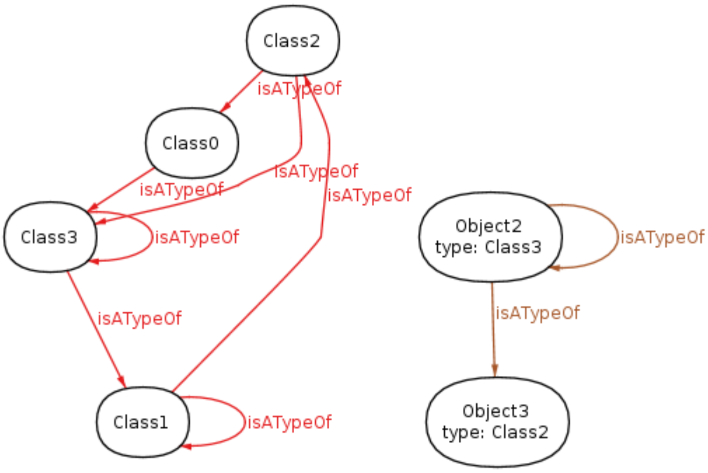

Fig. 9. Initial instance

```
one sig ObjectDiagram{
  -- as before }
  {
   -- valid relation
   all o1, o2: objects | o1->o2 in inheritance =>
    o1.type->o2.type in ClassDiagram.inheritance
  }
pred noSymmetric[r: univ->univ]{ no ~r & r }
pred noReflexive[r: univ->univ]{ no iden & r }
pred acyclic[r: univ->univ]{ no iden & ^r }
fact{
  let cdi = ClassDiagram.inheritance | {
    noReflexive[cdi]
    acyclic[cdi]
    noSymmetric[cdi]
    }
  }
```

Fig. 10. Corrections for the excerpt of the meta-model

thought Alloy does not have recursive functions, the transitivity closure can be used to iterate over a relation. The author of this paper believes that the simple but powerful notation of Alloy makes it possible to build software models without ambiguities. The author also believes that the specification obtained from the application of Alloy, will be a great requirements document for the implementation of an object and class diagram modeling tool.

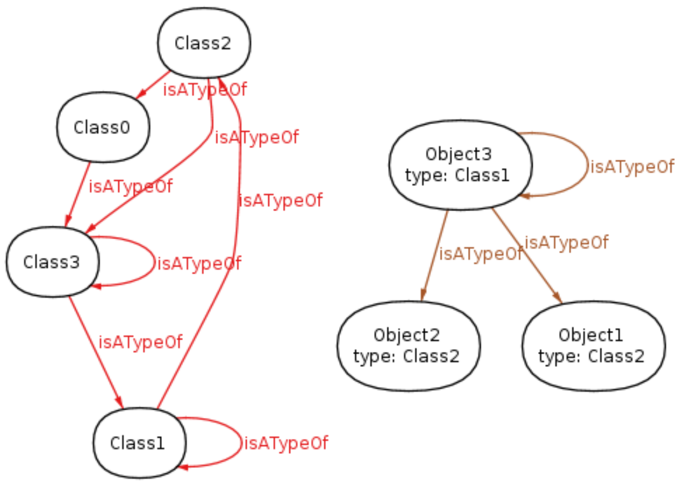

Fig. 11. Second instance

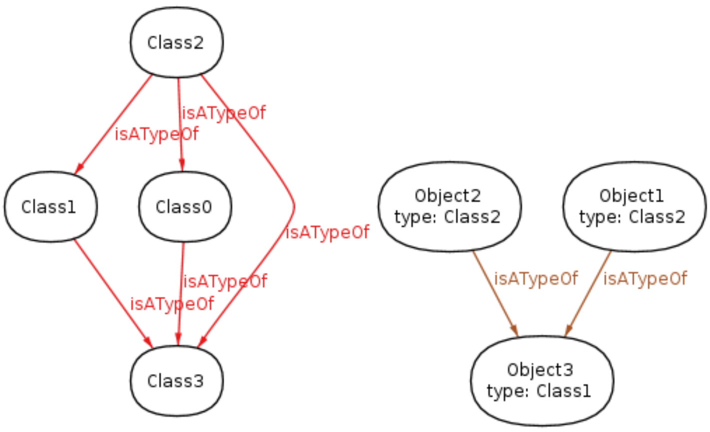

Fig. 12. Third instance

References

1. UML 2.0 superstructure specification. Tech. rep., Object Management Group (OMG) (2005)
2. Amalio, N., Polack, F.: Comparison of formalization approaches of UML class constructs in Z and Object-Z. In: ZB 2003: Formal Specification and Development in Z and B. Springer-Verlag (2003)
3. Ambler, S.W.: The Elements of UML 2.0 Style. Cambridge University Press (2005)
4. Aredo, D.B.: Formalizing UML class diagrams in PVS. In: Workshop on Rigorous Modeling and Analysis with the UML: Challenges and Limitations, at OOPSLA'99. Denver, Colorado, USA (1999)

5. Bennett, S., Skelton, J., Lunn, K.: UML. Schaums outlines, USA (2005)
6. Berardi, D., Calvanese, D., Giacomo, G.D.: Reasoning on UML class diagrams using description logic based systems. In: Proc. of the KI2001 Workshop on Applications of Description Logics. CEUR Electronic Workshop Proceedings (2001)
7. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide. Addison-Wesley Professional, 2^{nd} edn. (2005)
8. Brucker, A.D., Wolff, B.: A proposal for a formal OCL semantics in Isabelle/HOL. Lecture Notes in Computer Science 2410(147-175) (2002)
9. Bruel, J., Henderson-Sellers, B., Barbier, F., Parc, A.L., France, R.: Improving the UML metamodel to rigorously specify aggregation and composition. In: 7^{th} International Conference on Object-Oriented Information Systems (OOIS '01) (2001)
10. Cali, A., Calvanese, D., Giacomo, G.D., Lenzerini, M.: A formal framework for reasoning on UML class diagrams. In: Proceedings of the 13^{th} International Symposium on Foundations of Intelligent Systems, LNCS 2366. Springer-Verlag (2002)
11. Clavel, M., Egea, M.: Equational specification of UML+OCL static class diagrams. Tech. rep. (2006)
12. Evans, A.S.: Reasoning with UML class diagrams. In: Proceedings of the Second IEEE Workshop on Industrial Strength Formal Specification Techniques. IEEE Computer Society, Washington, DC, USA (1998)
13. Fowler, M.: UML Distilled: A Brief Guide to the Standard Object Modeling Language. Addison-Wesley, 3^{rd} edn. (2003)
14. France, R.B., Bruel, J.M., Larrondo-Petrie, M.M., Shroff, M.: Exploring the semantics of UML type structures with Z. In: Proceedings of the Second IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'97) (1997)
15. Funes, A., George, C.: Formal fundations in RSL for UML class diagrams. Tech. rep., The United Nations University (2002)
16. Gogolla, M., Bohling, J., Richters, M.: Validating UML and OCL models in USE by automatic snapshot generation. Software and System Modeling 4(4), 386–398 (2005)
17. Hassan, W., Logrippo, L.: Governance Policies for Privacy Access Control and their Interactions. IOS Press, Canada (2005)
18. Jackson, D.: Software Abstractions: Logic, Language and Analysis. MIT Press, England (2006)
19. Jordi Cabot, R.C., Riera, D.: Verification of UML/OCL class diagrams using constraint programming. In: ICST Workshop on Model Driven Engineering, Verification and Validation: Integrating Verification and Validation in MDE (MoDeVVa'2008) (2008)
20. Kim, S.K., Carrington, D.: Formalizing the UML class diagram using Object-Z. In: 2^{nd} International Conference on The Unified Modeling Language. Springer, Fort Collins, Colorado (1999)
21. Kim, S.K., Carrington, D.: A formal mapping between UML models and Object-Z specifications. In: ZB 2000, LNCS 1878. Springer-Verlag (2000)
22. Kleppe, A., Rensink, A.: A graph-based semantics for UML class and object diagrams. Tech. Rep. TR-CTIT-08-06, Enschede (January 2008)
23. Kuske, S., Gogolla, M., Kollmann, R., Kreowski, H.J.: An integrated semantics for UML class, object and state diagrams based on graph transformation. In: Lecture Notes in Computer Science. pp. 11–28. Springer (2002)
24. Kyas, M., Fecher, H., de Boer, F.S., Jacob, J., Hooman, J., Zwaag, M.V.D., Arons, T., Kugler, H.: Formalizing UML models and OCL constraints in PVS. Electronic Notes in Theoretical Computer Science 115, 39–47 (January 2005)

25. Larman, C.: Applying UML and Patterns : An Introduction to Object-Oriented Analysis and Design and Iterative Development. Prentice Hall, USA, 3^{rd} edn. (2004)
26. Ledang, H., Souquiéres, J.: Modeling class operations in B: Application to UML behavioral diagrams. In: 16^{th} Annual International Conference on Automated Software Engineering, 2001 (ASE 2001) (2001)
27. Meng, S., Aichernig, B.K.: Towards a coalgebraic semantics of UML: Class diagrams and use cases. Tech. rep., The United Nations University (2003)
28. Nolte, T.: Exploring Filesystem Synchronization with Lightweight Modeling and Analysis. Ph.D. thesis, M.I.T., U.S.A. (2002)
29. Pender, T.: UML Bible. John Wiley & Sons (2003)
30. Pilone, B., Pitman, N.: UML 2.0 in a Nutshell. O'Reilly (2005)
31. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual. Addison-Wesley Professional, 2^{nd} edn. (2005)
32. Valles-Barajas, F.: A formal model for a requirements engineering tool. In: First Alloy Workshop, co-located with the 14^{th} ACM/SIGSOFT Symposium on Foundations of Software Engineering (FSE'06). ACM, Portland, Oregon (2006)
33. Valles-Barajas, F.: A formal model for a state machine modeling tool: A lightweight approach. In: 3rd IEEE Systems and Software Week (SASW 2007). IEEE/NASA, Baltimore, Maryland (2007)
34. Valles-Barajas, F.: A metamodel for the requirements diagrams of SysML. IEEE Latin America Transactions 8(3), 259–268 (2009)
35. Valles-Barajas, F.: Use of a lightweight formal method to model the static aspects of state machines. Journal of Innovations in Systems and Software Engineering (ISSE): A NASA Journal 5(4), 255–264 (2009)
36. Valles-Barajas, F.: A precise specification for the modeling of collaboratios. Malaysian Journal of Computer Science 23(1), 18–36 (2010)
37. Warmer, J., Kleppe, A.: The Object Constraint Language: Getting Your Models Ready for MDA. Addison-Wesley Professional, 2^{nd} edn. (2003)

Fernando Valles-Barajas obtained a graduate degree in Computer Science at Center for Research and Graduate He received an MS in Control Engineering (1997) and a PhD in Artificial Intelligence (2001) from Monterrey He was a research assistant at Mechatronics department of ITESM campus Monterrey (1997-2001).

He received certification as a PSP Developer from the Software Engineering Institute of Carnegie Mellon

He is member of the IEEE and ACM.

His research interests include topics in Software Engineering and Control Engineering.

Currently he is full-time professor in the Information Technology Department at Universidad Regiomontana, Monterrey, Nuevo León, México. He also teaches modules in Software Engineering.

Received: February 10, 2011; Accepted: August 29, 2011.