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...
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...
Using Lightweight Formal Methods to Model Class and Object Diagrams

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 an-
alyze 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 dia-
grams (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 develop-
ers 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 dia-
grams 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 model-
ing 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 pa-
per was later presented in [35]. In [34] a model of System Modeling Language
(SysML) requirements diagrams using Alloy is included. It is important to men-
tion that SysML is a UML profile.
UML collaborations model interactions between entities working together to ac-
complish 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 pro-
gramming 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 ob-
servers. 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 these 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 \( \ast \) 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 (\( \neg \)), and (\( \& \& \)), or (\( \mid \mid \)) and the bi-implication (\( \iff \)). 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: \( \forall c_1, c_2, c_3 : C \ (c_1 Rc_2 \& c_2 Rc_3) \Rightarrow not \ (c_1 Rc_3) \).

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 compo-
sition 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 (umlMeta-
models). 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 manip-
ulate relations. Line 6 defines a signature for class diagrams. As was mentioned
in section 2 the relations between entities (represented in the model as signa-
tures) are specified using fields of signatures; for instance, the classes field of
the signature ClassDiagram denotes a binary relation between the ClassDia-
gram 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 gener-
ated.

Some ternary relations are defined in the ClassDiagram signature; for ex-
ample the association field is a ternary relation of the form:
association: ClassDiagram→classes→classes. With the definition of this rela-
tion, a class diagram knows all classes that are related by the association rela-
tion. 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 rela-
tions are specified. In line 1 one function, which is a reusing expression, is de-
fined. 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
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.
Using Lightweight Formal Methods to Model Class and Object Diagrams

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

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

9 one sig ClassDiagram{
10   -- as before }
12 antiTransitive[dependency, classes]
13 }

14 one sig ObjectDiagram{
16   -- as before }
17 association in
18       validRelation[ClassDiagram.association]
19 dependency in
20       validRelation[ClassDiagram.dependency]
21 antiTransitive[dependency, objects]
22 }

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
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 }

-- 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.ˆinheritance) <= 5

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

-- 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]|
  cd != cd' 
  f ln cd.classes
  some ge: GeneralizationStructure |
  ge.superClass = f && ge.isIncomplete = True
  f->s not ln 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-
1 sig Object{
2   -- as before
3   features: some Feature,
4   classification: set Class,
5   multipleClassification: Bool
6   }
7 type in ClassDiagram.classes
8   -- models multiple classification
9   classification in getGeneralizationStructure{ type }.subClasses
10 #classification > 1 =>
11   multipleClassification in True
12
13   -- an object contains the features of
14   -- its type (class) and the ancestors of
15   -- this type
16   let i = ClassDiagram.inheritance |
17   -- * is the reflexive transitive closure operator
18   features =
19     { f: Feature | f in i.type.features }
20
21 fun getAncestors[c: Class]: set Class{
22   -- * is the transitive closure operator
23   { x: Class | x in `(ClassDiagram.inheritance).c }
24 }
25
26 fun getDescendants[c: Class]: set Class{
27   { x: Class | x in c.`(ClassDiagram.inheritance) }
28 }
29
30 fun getGeneralizationStructure{ c: Class }:
31   set GeneralizationStructure{
32     ge: GeneralizationStructure | ge.superClass = c }
33
34 pred mutateClassification{ o: Object, c: set Class }{
35   o.classification = c
36 }
37
38

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.
Using Lightweight Formal Methods to Model Class and Object Diagrams

1 one sig ClassDiagram{
2   -- as before }
3 {
4   -- as before
5   acyclic[aggregation, classes]
6   irreflexive[aggregation]
7   antisymmetric[aggregation]
8   acyclic[composition, classes]
9   irreflexive[composition]
10  antisymmetric[composition]
11 }

12 one sig ObjectDiagram{
13   -- as before }
14 {
15   -- as before
16   composition in
17   validRelation[ClassDiagram.composition]
18   aggregation in
19   validRelation[ClassDiagram.aggregation]
20   acyclic[aggregation, objects]
21   irreflexive[aggregation]
22   antisymmetric[aggregation]
23   acyclic[composition, objects]
24   irreflexive[composition]
25   antisymmetric[composition]
26 }

27 pred delObject| od, od’: ObjectDiagram, o: Object{
28   -- pre conditions
29   o in od.objects
30   not o in (od.aggregation).(od.objects)
31   -- post conditions
32   od’.objects = od.objects - o
33 }

34   -- this operation takes into account the
35   -- composition relation
36 pred addPart| od, od’: ObjectDiagram, w, p: Object{
37   -- pre conditions
38   (w + p) in od.objects
39   w->p in validRelation[ClassDiagram.composition]
40   not w->p in od.composition
41   -- no sharing
42   not p in od.composition[Object]
43   -- post conditions
44   od’.composition = od.composition + w->p
45 }

Fig. 6. The aggregation and composition relation
**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 rela-
tion was changed to \textit{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.

---

1 module umlMetamodels/classModel
2 sig Name{}
3 sig Class{ name: Name }
4 one sig ClassDiagram{
5 classes: some Class,
6 inheritance: classes->classes
7 }
8 sig Object{
9 name: lone Name,
10 type: Class
11 }
12 one sig ObjectDiagram{
13 objects: some Object,
14 inheritance: objects->objects
15 }

---

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
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.
Using Lightweight Formal Methods to Model Class and Object Diagrams

Fig. 11. Second instance

Fig. 12. Third instance

References

Using Lightweight Formal Methods to Model Class and Object Diagrams


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.