Towards A Formalization of UML Class Structures in Z

Malcolm Shroff and Robert B. France
Department of Computer Science & Engineering
Florida Atlantic University
Boca Raton, FL-33431, USA
{mshroff,robert}@cse.fau.edu

Abstract

There is much interest in developing a firm semantic basis for object-oriented modeling concepts. By providing precise characterizations of object-oriented (OO) modeling concepts one gains the ability to build precise OO models of behavior and structure that can be rigorously analyzed.

In this paper we present the current results of our ongoing formalization of the Unified Modeling Language (UML). UML is a proposed common OO modeling language, thus it is important that it have a formally defined semantic base. The focus of this paper is the formalization of the primary UML constructs used to build class structures. We use the Z notation to precisely express the meaning of UML class structures.

Keywords: Formal Specification Techniques, Integrated Methods, Object-Oriented Analysis and Modeling, Unified Modeling Language, Z.

1 Introduction

Most popular graphical object-oriented (OO) modeling techniques provide mechanisms that allow one to:

- abstract over details best left to later stages of development,
- model an application along different views, and
- modularize problems and solutions.

The static aspects of problems and solutions are modeled, in most OO methods, by imposing a structure that makes clear the major concepts and relationships that are pertinent to the required behavior of the system. The basic unit of structure is usually an abstractly defined type called a *class*. Extensionally, a class is a collection of instances that share common properties. Intensionally, a class is a characterization of a collection of instances that have common properties. Static models also include constructs for expressing relationships among classes, and for structuring the classes into groups that can be abstractly referred to as units.

It has been stated that the simple and graphical nature of OO modeling constructs facilitates the creation of concise and understandable models. Our experiences indicate that the lack of firm semantic foundations for OO modeling notations can make understanding OO models more apparent than real. In particular, the lack of adequate semantic foundations makes rigorous analysis of semantic properties captured by the models difficult.

In this paper we present a formal semantic base for the Unified Modeling Language (UML) [2] class structure constructs. UML is the proposed common language for OO modeling currently being developed by a group of expert OO method developers[1]. It is essentially a distillation of some of the best experiences reflected in major second-generation OO methods. As a proposed common language, it is important that the notation have a well-defined semantic base. In this paper we propose such a semantic base for the subset of UML notation used for creating class structures. We use the Z notation [16] to express the semantics of UML class structures. Z was used as the underlying formal notation because of its maturity as a formal specification notation, and because it is based on mathematical concepts that are well-known and understood (set theory, and predicate logic). Furthermore, the schema constructs of Z can be directly linked to the concepts of classes, and class structures, providing a clear link between OO constructs and their formally expressed interpretations. The use of a mathematically-based notation such as Z also means that it is possible to formally assess the soundness of class structures.

The semantic base that we propose in this paper is based on our experiences with formalizing Fusion models (e.g., see [1, 4, 5, 9]), the well-defined OO concepts of Syntropy [8], and the Z formalization of class structures given in Hall [10] and Hammond [11]. The work presented in this paper differs from these works

[1] See the WWW site at http://www.rational.com/ot/uml.html

in that it is targeted specifically to UML.

In Section 2 we give an overview of the UML and Z notations. In Section 3 we present our formalization of UML class structuring constructs. We conclude in Section 4 with a summary of work done and an outline of our planned work in this area.

2 An Overview of UML and Z

In this section we present brief descriptions of the UML and Z notations.

2.1 UML - An Overview

The Unified Modeling Language (UML) represents, primarily, the unification of the Booch and OMT methods. It provides the basis for a *de facto* standard for specifying, visualizing and documenting the artifacts of an object-oriented system under development. In this section we give an overview of the notation used to build class structures. A class structure is depicted as a Class Diagram in UML.

A class in UML is represented as a rectangular box with three compartments. The top compartment contains the class name, the middle one contains a list of attributes (with optional types and initial values), and a list of operations (with optional argument lists and return types) are shown in the last compartment. In this paper we model only the static aspects of classes (i.e., we ignore the operations).

Associations represent structured relationships between objects of different classes, and is depicted as a line connecting classes. An association may have a name with an optional directional arrow showing the direction of the named relationship. Each end of an association may have a role name showing how the class is viewed by the other class in the association. Multiplicity of an association represents how many instances of the class can be associated with one instance of the other class. A range is indicated by an integer (the lower value), two dots, and an integer (the upper value); a single integer is a valid range and the symbol '*' indicates many i.e. an unlimited number of objects. By itself, the symbol '*' is equivalent to '0..*' i.e. any number including none.

Aggregation is a special form of association that represents the whole-part relationship between objects. It is indicated by placing a diamond on the end of the association attached to the whole object of the whole-part relationship.

Generalization is a powerful tool for abstraction. It highlights the similarities and differences between the classes. The class at the root of the hierarchy (the superclass) captures the properties that are common to all the other classes in the hierarchy (the subclasses). As one goes down the hierarchy, the classes become more specialized versions of the higher level classes. All attributes, operations and associations of the superclass are inherited by the subclasses.

Examples of UML class structure constructs are given throughout the paper.

2.2 Z - An Overview

Z [16] is a formal specification notation created by J.-R. Abrial and further developed by the Programming Research Group at Oxford[2]. It is based on the well-known mathematical concepts of set theory and predicate logic.

The primary construct in Z is the *schema*. A schema is divided into two parts: a declaration part and a predicate part. The declaration part consists of declarations of the form $v : T$, where v is a variable of type T. This declaration means that the value of v is a member of set T (types are sets in Z). The predicate part consists of predicate logic expressions that define the relationships between the declared variables. A Z schema has the following form:

SchemaName
Declaration
Predicate

Types in Z can be either *basic* or *composite*. The elements of the basic types (or given sets) are used as building blocks for more complex composite types. A basic type is presented as follows:

$[BASIC\_TYPE\_NAME]$

[2] See the WWW site at http://www.comlab.ox.ac.uk/archive/z.html

There are three kinds of composite types in Z: set types, cartesian product types and schema types. Examples of composite types are given in the following schema:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Example}} \\ \hline
s : \mathbb{P}\, Person \\
t : A \times B \\
u : Schema \\ \hline
\end{array}$$

s is a set of elements from $Person$. t is a pair of elements in which the first element is an element of A and the second an element of B. u is a binding of values to the variables declared in schema $Schema$.

Schemas are used to model both the static and dynamic aspects of systems. State schemas capture the static aspect of a system, whereas operation schemas capture dynamic aspects.

In a state schema, the components of the system's state are declared in the declaration section and the constraints on the state are declared in the predicate part. The following example shows the declaration of the state schema for a phone database:

$[Person, Phone]$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{PhoneDB}} \\ \hline
members : \mathbb{P}\, Person \\
telephones : Person \leftrightarrow Phone \\ \hline
\operatorname{dom} telephones \subseteq members \\ \hline
\end{array}$$

$Person$ and $Phone$ are the two given sets of the system. $PhoneDB$ describes the state of the system. A state consists of two sets: $members$ (set of subscribers) and $telephones$ (set of pairs representing the relation that exists between persons and their telephones). The predicate part states that only persons in $members$ (the subscribers) can be associated with telephones in the system.

An operation schema defines an operation in terms of the relationship between the state before the operation executes and the state after it has completed execution. The declaration part contains variables representing the before and after states, inputs and outputs. The predicate part of the schema defines the relationship between the operation's before and after states. The following conventions are used:

- Unprimed variables (eg. v) - value of a variable before execution of operation.
- Primed variable (eg. v') - value of a variable after execution of the operation.
- Variable ending in '*?*' - an input to an operation.

- Variable ending in '!' - an output from an operation.
- ΔS - denotes a change in the state S.
- ΞS - denotes that the state S does not change after the operation.

The operation to add an entry to the phone database changes the phone database (*PhoneDB*). The input to the operation is the name of the subscriber and the output is the new telephone number for that subscriber. The precondition is that the input name (*name*?) belongs to *members* and that the new number (*newnumber*?) does not already exist in the database. At the end of the operation, the new number has been added to *telephones* and the set *members* is not changed. The Z schema for this operation is shown below:

$$
\begin{array}{|l}
\hline
\textit{AddEntry} \\
\Delta \textit{PhoneDB} \\
\textit{name}? : \textit{Person} \\
\textit{newnumber}? : \textit{Phone} \\
\hline
\textit{name}? \in \textit{members} \\
\textit{name}? \mapsto \textit{newnumber}? \notin \textit{telephones} \\
\textit{telephones}' = \textit{telephones} \cup \{\textit{name}? \mapsto \textit{newnumber}?\} \\
\textit{members}' = \textit{members} \\
\hline
\end{array}
$$

3 Formalizing UML Class Structures

In this section we present our Z formalization of UML class structuring constructs. Specifically, we formally express, in Z, the notions of a class, class instance, and class associations, and provide formalizations of aggregation and generalization structures. We illustrate the formalizations with examples, most of which are taken from the UML manual Ver. 0.8 [2].

3.1 Representing Classes in Z

The static aspect of a class (i.e., its attributes and object identifier values) is formalized by a Z state schema called the *Class Schema*. The attributes and object identifiers of class instances are represented by state variables. Any invariant associated with the class, usually expressed as annotations in a Class Diagram, are expressed formally in the predicate part of the Z class schema. The type name of the attribute in the class schema corresponds to the type name of the attribute in the class. The type must also be defined in Z as a basic type or a schema. If no type is associated with an attribute, then the capitalized name of the attribute is used as the type name in the Z class schema and declared as a Z basic type.

The bindings of values to variables declared in the Z class schemas represent instances of the class. The object identifier of an instance is the value bound to the object identifier variable declared in the Z class schema.

As an example of the Z formalization of the static aspect of a class, consider the class *Polygon* with attributes *center*, *vertices*, *bordercolor* and *fillcolor* as shown in Fig. 1.

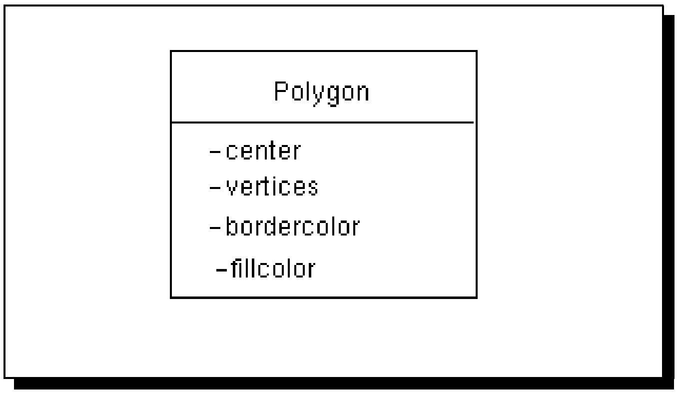

Figure 1: Polygon Class

The class schema for the polygon class is as follows:

$[\mathit{Polygon\_ID}, \mathit{CENTER}, \mathit{VERTICES}, \mathit{BORDERCOLOR}, \mathit{FILLCOLOR}]$

Polygon

$$\begin{array}{ll}
\mathit{ident} : \mathit{Polygon_ID} & [\mathit{Object\ \ Identifier}] \\
\mathit{center} : \mathit{CENTER} & \\
\mathit{vertices} : \mathit{VERTICES} & \\
\mathit{bordercolor} : \mathit{BORDERCOLOR} & \\
\mathit{fillcolor} : \mathit{FILLCOLOR} &
\end{array}$$

An instance of the class *Polygon* is identified by the object identifier *ident*, which is of type *Polygon_ID*.

3.2 Representing Associations in Z

UML defines associations as structural relationships between classes. An instance of an association is a link, which is a tuple of object references. In a binary association only two object are involved. Ternary associations are difficult to implement and are generally avoided.

We adapted the FuZE mathematical toolkit [4] we created for the common types of binary relationships that can occur in a Fusion [7] class structure (for example, one-to-one, one-to-many, many-to-many) to UML binary associations. Graphical symbols are used to represent associations in the toolkit. In Fig. 2 an extract from the mathematical toolkit is given. The schema in Fig. 2 states that a many-to-many UML

$$
\begin{array}{l}
[X, Y] \\
\hline
* \longleftrightarrow * : \mathbb{P}(X \leftrightarrow Y) \\
1 \longleftrightarrow 1 : \mathbb{P}(X \rightarrowtail\!\!\!\!\rightarrow Y) \\
\ldots
\end{array}
$$

Figure 2: An extract from the Mathematical Toolkit

association is a mathematical relation, and a 1-to-1 UML association is a bijection.

Fig. 3 shows a binary association between a *Company* class and a *Person* class. The multiplicity of the

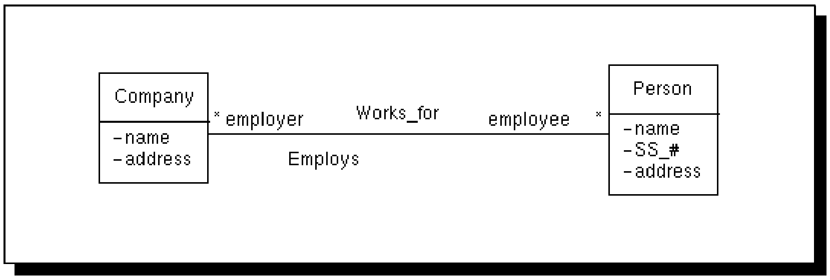

Figure 3: Example of a binary relationship.

association states that a company may employ many persons and a person may work for many companies. The association *Works_for* between *Person* and *Company* is defined, using the toolkit definition, as follows:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\ \mathit{Works\_for}\ }}\\
company : \mathbb{P}\, Company \\
person : \mathbb{P}\, Person \\
works\_for : Person \leftrightarrow Company \\
\hline
works\_for \in * \longleftrightarrow *[person, company] \\
\forall\, i, j : company \bullet (i.ident = j.ident \Leftrightarrow i = j) \\
\forall\, i, j : person \bullet (i.ident = j.ident \Leftrightarrow i = j) \\
\hline
\end{array}
$$

In the schema *Works_for*, the sets *company* and *person* are the current sets of *Company* and *Person* instances respectively, that is, they consist of instances that have been created but not yet destroyed in the system being modeled. The first predicate in the schema states that the association is many-to-many (there is no need for this predicate in the many-to-many case; we include it here only to show how the associations defined in the mathematical toolkit are used), and the remaining predicates state that object identifiers are unique.

Role names are simply names for the domain and range of the associations they are attached to. The Z schema for the association *Employs* that refers to role names is given below:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\ \mathit{Employs}\ }}\\
employer : \mathbb{P}\, Company \\
employee : \mathbb{P}\, Person \\
employs : Company \leftrightarrow Person \\
\hline
\mathrm{dom}\, employs = employer \\
\mathrm{ran}\, employs = employee \\
\forall\, i, j : employer \bullet (i.ident = j.ident \Leftrightarrow i = j) \\
\forall\, i, j : employee \bullet (i.ident = j.ident \Leftrightarrow i = j) \\
\hline
\end{array}
$$

Fig. 4 shows an association with attributes (link attributes), and with cardinalities not supported by the toolkit.

The Z formalization of the association is given below:

$[SEMESTER]$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\ \mathit{StAttrib}\ }}\\
semester : SEMESTER \\
\hline
\end{array}
$$

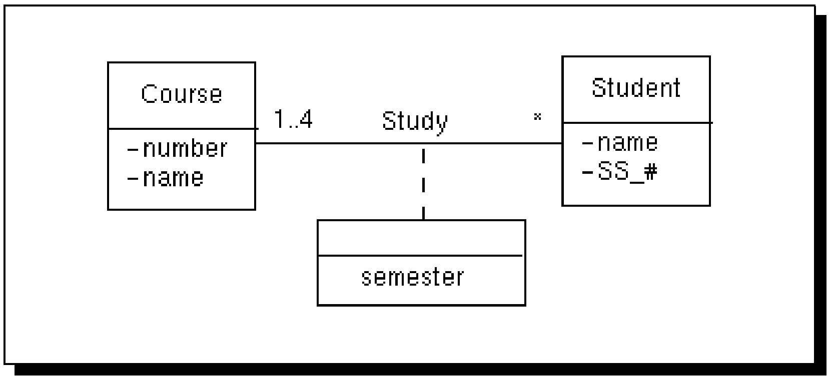

Figure 4: An association with attributes.

$$
\begin{array}{|l}
\textit{Study} \\
\hline
course : \mathbb{P}\, Course \\
student : \mathbb{P}\, Student \\
study : Student \leftrightarrow Course \\
study\_ \ attrib : (Student \times Course) \rightarrow StAttrib \\
\hline
\operatorname{dom} study = student \\
\operatorname{ran} study \subseteq course \\
\operatorname{dom} study\_attrib = study \\
\forall\, s : student \bullet 1 \leq \#study(\!|\, \{s\} \,|\!) \leq 4 \\
\forall\, i, j : student \bullet (i.ident = j.ident \Leftrightarrow i = j) \\
\forall\, i, j : course \bullet (i.ident = j.ident \Leftrightarrow i = j) \\
\hline
\end{array}
$$

The first two predicates specify the domain and range of the association (i.e., the instances that are linked). The third predicate links the domain of *study_attrib* to *study* and the fourth predicate expresses the cardinality constraint. The Z expression $R(\!|\, \{s\} \,|\!)$ returns the set of elements that are related to s in the relation R, and the $\#$ operator returns the number of elements in a set. The remaining predicates state that the object identifiers are unique.

3.3 Representing Aggregation in Z

In UML, aggregation is a special type of association that indicates a lifetime dependency among the related parts. One end of the association is considered the 'whole' part of the aggregate structure and the other ends are considered 'parts' of the class at the 'whole' end. The class at the whole end is referred to as the *aggregate class*, and the classes at the parts end are *part classes*. Instances of aggregate classes (aggregate instances) can be thought of as structures in which instances of parts classes (part instances, or, simply, parts) are components. Two types of aggregation are distinguishable in UML: *physical* and *catalog* aggregation. In a physical aggregation a part instance can belong to only one aggregate instance (no sharing of parts across aggregate instances allowed). There is no such restriction for catalog aggregation, that is, parts can be shared across catalog aggregate instances.

Fig. 5 shows a physical aggregation in which *Window* is an aggregate class with part classes *Title Bar* and *Pane*. *Title Bar* is also an aggregate class with part classes *Close Button* and *Title*. Fig. 6 shows

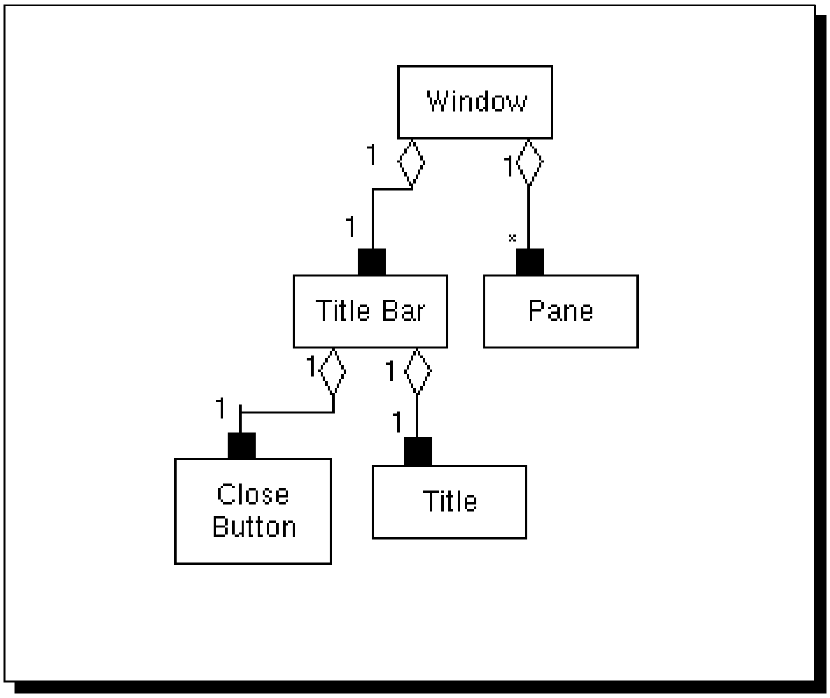

Figure 5: A UML physical aggregation structure.

a catalog aggregation structure in which *Company* is the aggregate class with a *Division* part class. An instance of *Division* can be shared across *Company* aggregate instances.

The UML manual states "the lifetimes of the parts are dependent on the lifetime of the whole", but it does not make clear precisely what the dependency is. Furthermore, the UML description of aggregation

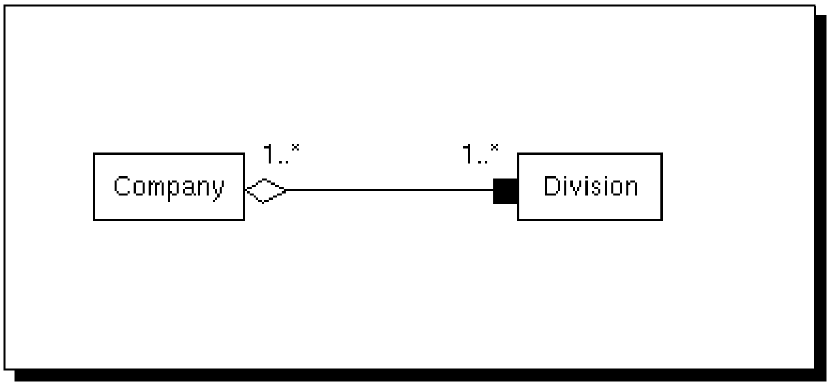

Figure 6: A UML catalog aggregation structure.

does not make clear whether instances of part classes may or may not exist outside of an aggregate structure. We based our formalization of UML aggregation on the following:

- Aggregation associations are mandatory, unless they have a multiplicity that indicates optionality (e.g., $*$). Aggregation associations are similar to general associations in this respect. For example, in Fig. 5, all instances of the part classes (e.g., instances of *Pane*) must be parts of aggregate instances (e.g., instances of *Window*) because their multiplicities (1) with respect to the aggregate classes do not reflect optionality.

- Lifetime dependency means that when parts of an aggregate instance are deleted such that the multiplicity constraints of the aggregate class structure are violated, then the whole and all its parts are destroyed for physical aggregation, and for catalog aggregation, the whole and the parts that are not part of any other existing aggregate instances are destroyed.

The Z formalization of the aggregate structure in Fig. 5 is given below:

$[CloseButton\_ID, Title\_ID, TitleBar\_ID, Pane\_ID, Window\_ID]$

CloseButton

$ident : CloseButton\_ID$

Title

$ident : Title\_ID$

$$
\begin{array}{|l}
\hline
\textit{TitleBar} \\
ident : TitleBar\_ID \\
closeButton : CloseButton \\
title : Title \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{Pane} \\
ident : Pane\_ID \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{Window} \\
ident : Window\_ID \\
titleBar : TitleBar \\
panes : \mathbb{P}\, Pane \\
\hline
\forall i, j : panes \bullet i.ident = j.ident \Leftrightarrow i = j \\
\hline
\end{array}
$$

The above formalizes only the structure of the aggregate object; it does not, for example, stipulate that a part instance appears in only one aggregate instance. This constraint, among others, is specified in the Z schema representing the *system (instance) view* of the structure. In the system view, the structure is viewed in terms of the instances that have been created in the system and not yet destroyed. For example, the system view of the *Window* aggregation structure consists of the current instances of *Window*, *TitleBar*, *Pane*, *CloseButton*, and *Title*, that exist within the system.

The Z formalization of the *Window* aggregation system view is given below:

$$
\begin{array}{|l}
\hline
\textit{ClassStruct1} \\
windows : \mathbb{P}\, Window \\
buttons : \mathbb{P}\, CloseButton \\
titles : \mathbb{P}\, Title \\
titlebars : \mathbb{P}\, TitleBar \\
panes : \mathbb{P}\, Pane \\
\hline
\forall w1, w2 : windows \mid w1.titleBar = w2.titleBar \lor \\
\quad (w1.panes \neq \varnothing \land w1.panes = w2.panes) \bullet w1 = w2 \\
\forall w : windows \bullet w.titleBar \in titlebars \land w.panes \subseteq panes \\
\forall t : titlebars \bullet t.closeButton \in buttons \land t.title \in titles \\
\forall tb : titleBars \bullet (\exists w : windows \bullet w.titleBar = tb) \\
\forall p : panes \bullet (\exists w : windows \bullet p \in w.panes) \\
\forall b : buttons \bullet (\exists tb : titlebars \bullet tb.closeButton = b) \\
\forall t : titles \bullet (\exists tb : titlebars \bullet tb.title = t) \\
\forall w1, w2 : windows \bullet w1.ident = w2.ident \Leftrightarrow w1 = w2 \\
\forall b1, b2 : buttons \bullet b1.ident = b2.ident \Leftrightarrow b1 = b2 \\
\ldots \\
\hline
\end{array}
$$

In the schema *ClassStruct*1, *windows* is the set of *Window* instances that have been created but not yet destroyed in the system. The other declared sets represent similar sets of instances. The first predicate in the predicate part of the Z schema states that two elements of *windows* are the same instance of *Window* if they have at least one part that is common. The next two predicates state that all parts of *Window* and *TitleBar* instances must come from the sets of existing instances. The next four predicates in the schema state that all created instances of the part classes must be parts of created aggregate instances. The remaining predicates specify the uniqueness of object identifiers.

The Z formalization of the catalog composition shown in Fig. 6 is given below:

$[Company\_ID, Division\_ID]$

Division

$ident : Division\_ID$

Company

$ident : Company\_ID$
$divisions : \mathbb{P}\, Division$

$\# divisions \geq 1$

The fact that instances of *Division* can be shared among instances of *Company* can be obtained from the formalization of the structure's system view given below:

ClassStruct2

$companies : \mathbb{P}\, Company$
$divisions : \mathbb{P}\, Division$

$\forall\, d : divisions \bullet (\exists\, c : companies \bullet d \in c.divisions)$
$\forall\, c1, c2 : companies \bullet c1.ident = c2.ident \Leftrightarrow c1 = c2$
$\forall\, b1, b2 : divisions \bullet b1.ident = b2.ident \Leftrightarrow b1 = b2$

The only restrictions that the above schema places on created instances is that created instances of *Division* must be parts of created *Company* instances, and that the identifiers of instances be unique. This means that instances of *Division* are free to be parts of one or more created *Company* instances.

3.4 Representing Generalization Structures in Z

Generally, subclasses of superclasses in a generalization hierarchy are disjoint but sometimes they can overlap. Overlapping subclasses are indicated in UML by multiple inheritance structures.

In our Z formalization of generalization structures, the superclass is formalized in the same way as any other class. Specializations (subclasses) of a class are considered to be subspaces of the superclass instance space, and are formally defined by Z state schemas in which a variable of the superclass type is declared in the declaration part of the subclass, along with the variables for the other attributes of the subclass that are not attributes of the superclass. Fig. 7 shows a generalization/specialization relationship in which the subclasses *LandVehicle* and *WaterVehicle* inherit from the superclass *Vehicle*. The class *AmphibiousVehicle* is an example of multiple inheritance.

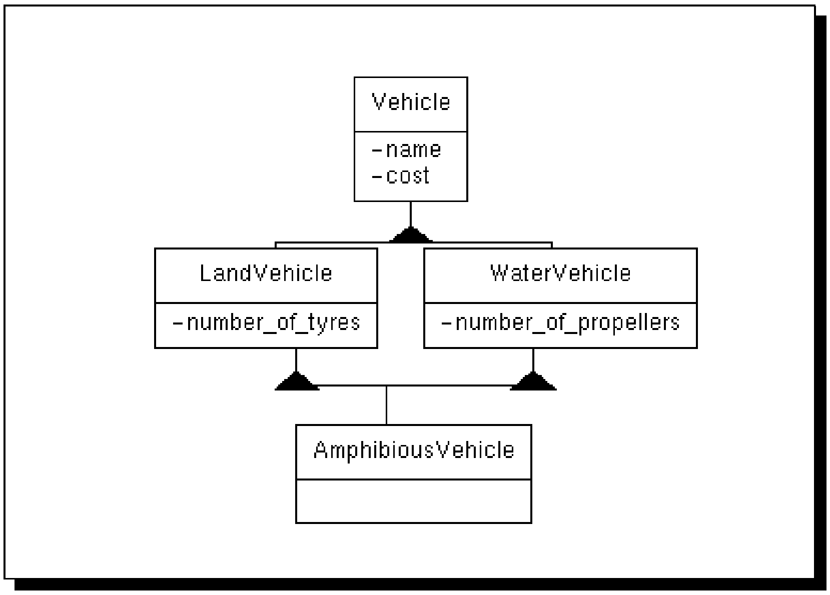

Figure 7: A generalization/specialization hierarchy.

The structural aspect of the *Vehicle* hierarchy is formalized below:

$[Vehicle\_ID, NAME, COST, N]$

Vehicle

$ident : Vehicle\_ID$
$name : NAME$
$cost : COST$

LandVehicle

$vehicle : Vehicle$
$number\_of\_tyres : \mathbb{N}$

$$
\begin{array}{|l}
\hline
\textit{WaterVehicle} \\
\hline
vehicle : Vehicle \\
number\_of\_propellers : \mathbb{N} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{AmphibiousVehicle} \\
\hline
landVehicle : LandVehicle \\
waterVehicle : WaterVehicle \\
\hline
landVehicle.vehicle = waterVehicle.vehicle \\
\hline
\end{array}
$$

The above schemas do not provide information needed to answer questions such as:

- Are there instances of *Vehicle* that are not instances of the shown subclasses?
- Are all instances of vehicles that are instances of both *LandVehicle* and *WaterVehicle* instances of *AmphibiousVehicle*?

The restrictions related to these questions need to be specified in the schema that formalizes the system view of the class structure.

Consider the following restrictions for the *Vehicle* hierarchy:

- There are instances of *Vehicle* that are not instances of the shown subclasses.
- All instances of *Vehicle* that are instances of both subclasses are instances of *AmphibiousVehicle*

The formalization of the system view of the *Vehicle* hierarchy under such restrictions is given below:

$$
\begin{array}{|l}
\hline
\textit{ClassStruct3} \\
\hline
vehicles : \mathbb{P}\, Vehicle \\
lvehicles : \mathbb{P}\, LandVehicle \\
wvehicles : \mathbb{P}\, WaterVehicle \\
avehicles : \mathbb{P}\, AmphibiousVehicle \\
\hline
\forall v : lvehicles \bullet v.vehicle \in vehicles \\
\forall v : wvehicles \bullet v.vehicle \in vehicles \\
\forall av : avehicles \bullet av.landVehicle \in lvehicles \land av.waterVehicle \in wvehicles \\
\forall lv : lvehicles;\ wv : wvehicles \mid lv.vehicle = wv.vehicle \bullet \\
\quad (\exists av : avehicles \bullet av.landVehicle = lv \land av.waterVehicle = wv) \\
\forall i, j : vehicles \bullet i.ident = j.ident \Leftrightarrow i = j \\
\hline
\end{array}
$$

The first two predicates of *ClassStruct*3 stipulate that created instances of land and water vehicles must be created instances of *vehicles*. The third predicate states that created instances of amphibious vehicles must be created instances of land and water vehicles. The fourth predicate expresses the restriction that all vehicles that are both land and water vehicles are amphibious vehicles. The remaining predicate expresses the uniqueness of object identifiers.

3.5 Formalizing A Complete Analysis Model

In this section, we present the Z formalization of a small Class Diagram. The formalization was done according to the guidelines given in the previous sections. Fig. 8 shows part of a Class Diagram taken from [15].

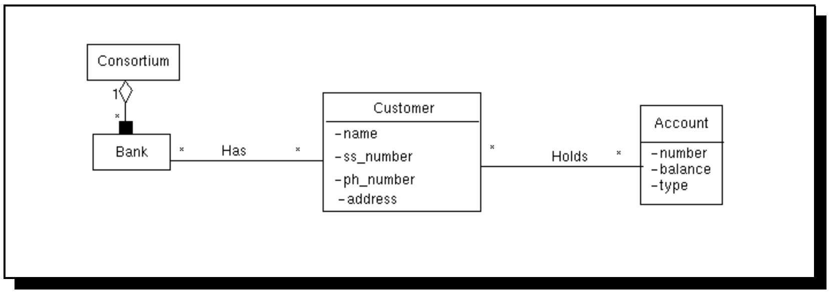

Figure 8: Class diagram of a small banking system.

Formalization proceeds first by formalizing the parts of the Class Diagram and then composing them in a Z schema formally characterizing the system view of the Class Diagram. The final schema obtained is named *Init*. In *Init*, invariants involving parts that were separately defined are expressed in the predicate part of the schema.

The Z formalization of the Class Diagram is given below:

$[Bank\_ID, Consortium\_ID, Customer\_ID, Account\_ID]$
$[NAME, SS\_NUMBER, ADDRESS, PH\_NUMBER, NUMBER, BALANCE, TYPE]$

Bank

$ident : Bank\_ID$

Consortium

$ident : Consortium\_ID$
$bank : \mathbb{P}\, Bank$

$\forall\, i, j : bank \bullet i.ident = j.ident \Leftrightarrow i = j$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Customer}} \\ \hline
ident : Customer_ID \\
name : NAME \\
ss_number : SS_NUMBER \\
address : ADDRESS \\
ph_number : PH_NUMBER \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Account}} \\ \hline
ident : Account_ID \\
number : NUMBER \\
balance : BALANCE \\
type : TYPE \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Has}} \\ \hline
bank : \mathbb{P}\, Bank \\
customer : \mathbb{P}\, Customer \\
has : Bank \leftrightarrow Customer \\ \hline
\forall\, i, j : bank \bullet (i.ident = j.ident \Leftrightarrow i = j) \\
\forall\, i, j : customer \bullet (i.ident = j.ident \Leftrightarrow i = j) \\
has \in * \longleftrightarrow *[bank, customer] \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Holds}} \\ \hline
customer : \mathbb{P}\, Customer \\
account : \mathbb{P}\, Account \\
holds : Customer \leftrightarrow Account \\ \hline
\forall\, i, j : account \bullet (i.ident = j.ident \Leftrightarrow i = j) \\
\forall\, i, j : customer \bullet (i.ident = j.ident \Leftrightarrow i = j) \\
holds \in * \longleftrightarrow *[customer, account] \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Init}} \\ \hline
Has \\
Holds \\
consortium : \mathbb{P}\, Consortium \\ \hline
\forall\, c1, c2 : consortium \mid c1.bank \neq \varnothing \wedge c1.bank = c2.bank \bullet c1 = c2 \\
\forall\, c : consortium \bullet c.bank \in bank \\
\forall\, b : bank \bullet (\exists\, c : consortium \bullet c.bank = b) \\
\forall\, i, j : consortium \bullet (i.ident = j.ident \Leftrightarrow i = j) \\ \hline
\end{array}$$

3.6 Formalizing the UML Operation Model

The Operation Specification (OS) of UML specifies the behavior of system operations. In this section we present an example of an OS formalization.

Consider the operation to borrow books in a library. Its Operation Specification is as follows:

Operation : CheckOutCopy

Responsibilities : This operation allows a registered user of the library to borrow a book.

Inputs : *reg* - a registered user of the library; *vol* - a copy of the book.

Returns : none

Modified Objects : *LibraryState*

Precondition : The following must hold

Only registered users can borrow books.

Only available copies of the book can be borrowed.

A borrower can only borrow books if the number

of loaned copies is less than the maximum number allowed.

Postcondition : The copy is removed from the list of available copies.

The borrower has a new copy on loan.

To shorten the presentation of the OS formalization, we omit the definition of the library state schema *LibraryState*. The Z formalization of the above Operation Specification is given below:

CheckOutCopy

$$\begin{array}{l}
\Delta LibraryState \\
reg? : Person \\
vol? : Copy \\
\hline
reg? \in borrower \\
vol? \in available \\
\# borrowedby(\!|\, \{reg?\} \,|\!) < MaxCopiesAllowed \\
available' = available \setminus \{vol?\} \\
checkedout' = checkedout \cup \{vol?\} \\
borrowedby' = borrowed \cup \{vol? \mapsto reg?\} \\
borrower' = borrower
\end{array}$$

The first three predicates of *CheckOutCopy* give the precondition of the operation, and the remaining predicates specify what happens when a copy has been successfully borrowed. Note that the operation schema only represents the successful operation of borrowing a book. More schemas are need for handing exceptions and error conditions.

4 Conclusion

Over the last year the *Methods Integration Research Group*[3] (MIRG) at Florida Atlantic University has worked on integrating formal methods with informal graphical object-oriented techniques (see, for example, [6, 9, 5]). The initial work concentrated on formalizing the Fusion object-oriented method [7]. A prototype tool called FuZe (Fusion/Z Environment) [4] was developed to implement the transformation rules. The tool was developed using the scripting language of Paradigm Plus [14] and was integrated with the Z type checker (ZTC) [13]. These transformation rules are now being extended to cover both the static and dynamic models of UML. This paper presents the current results of this work. Work is ongoing on the formalization of other class constructs (e.g., composites, and recursive structures) and on the formalization of UML dynamic models.

The primary purpose of formal methods is to help engineers construct more reliable systems. Graphical object-oriented models have the inherent advantage of closely reflecting real-world concepts. The primary objective of integrating UML and Z is to create machine analyzable UML models through the generation of Z specifications. The integration allows one to use Z type checkers (e.g., ZTC) and Z animation tools (e.g., ZANS [12]) to analyze the models. The development of automated tools to cover both the analysis and the design phases of object-oriented development, will enable rigorous consistency and refinement checks that are not possible with the sole use of the graphical UML modeling techniques. The rigorous, graphical OO-based development environment that we are developing will facilitate the rigorous specification and analysis of complex, safety-critical software systems.

References

[1] B. W. Bates, J.-M. Bruel, R. B. France, and M. M. Larrondo-Petrie. Formalizing Fusion Object-Oriented Analysis Models. In E. Najm and J.-B. Stephani, editors, ***Proceedings of the First IFIP International Workshop on Formal Methods for Open Object-based Distributed Systems,*** Paris, France. Chapman & Hall, London, UK, 4–6 Mar. 1996.

[2] G. Booch and J. Rumbaugh. Unified Method. Version 0.8, Rational Software Corporation, Santa Clara, CA-95051, USA, Oct. 1995.

[3] See WWW site at http://www.cse.fau.edu/research/MIRG

[3] J. P. Bowen and J. A. Hall, editors. ***Z User Workshop, Cambridge 1994***, Workshops in Computing. Springer-Verlag, New York, 1994.

[4] J.-M. Bruel, B. Chintapally, R. B. France, and G. K. Raghavan. FuZE–Draft of the User's Guide. FAU Technical Report TR-CSE-96-9, Department of Computer Science & Engineering, Florida Atlantic University, Boca Raton, FL–33431, USA, Feb. 1996.

[5] J.-M. Bruel, R. B. France, B. Chintapally, and G. K. Raghavan. A Tool for Rigorous Analysis of Object Models. In ***Proceedings of the*** 20^th ***International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'96),*** Santa Barbara, California, July 29–August 2 1996.

[6] J.-M. Bruel, R. B. France, M. M. Larrondo-Petrie, B. Chintapally, and G. K. Raghavan. CASE-based Rigorous Object-Oriented Modeling. In ***Proceedings of the Northern Formal Methods Workshop,*** Bradford, UK, 23–24 Sept. 1996.

[7] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. ***Object-Oriented Development: The Fusion Method.*** Prentice Hall, 1994.

[8] S. Cook and J. Daniels. Let's get formal. ***Journal of Object-Oriented Programming (JOOP)***, pages 22–24 and 64–66, July–Aug. 1994.

[9] R. B. France, J.-M. Bruel, and M. M. Larrondo-Petrie. An Integrated Object-Oriented and Formal Modeling Environment. ***To appear in the Journal of Object-Oriented Programming (JOOP)***, 1996.

[10] J. A. Hall. Specifying and Interpreting Class Hierarchies in Z. In Bowen and Hall [3], pages 120–138.

[11] J. A. R. Hammond. Producing Z Specifications from Object-Oriented Analysis. In Bowen and Hall [3], pages 316–336.

[12] X. Jia. ***An Approach to Animating Z Specifications.*** Division of Software Engineering, School of Computer Science, Telecommunication, and Information Systems, DePaul University, Chicago, IL 60604, USA, 1995. ZANS is a Z animator available on Internet via anonymous ftp at `ise.cs.depaul.edu`.

[13] X. Jia. ***ZTC: A Z Type Checker, User's Guide, version 2.01.*** Division of Software Engineering, School of Computer Science, Telecommunication, and Information Systems, DePaul University, Chicago, IL 60604, USA, May 1995. The ZTC tool and documentation are available on Internet via anonymous ftp at `ise.cs.depaul.edu`.

[14] Protosoft, Inc. ***Paradigm Plus – Release 2.0.*** Protosoft Technical Support, Houston, Texas, USA, Jan. 1994.

[15] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. ***Object-Oriented Modeling and Design.*** Prentice Hall, 1991.

[16] J. M. Spivey. ***The Z Notation: A Reference Manual.*** Prentice Hall, Englewood Cliffs, NJ, Second edition, 1992.