A Problem-Oriented Analysis of Basic UML Static Requirements Modeling Concepts

Robert B. France
Department of Computer Science
Colorado State University
Fort Collins, CO 80523, USA.
Tel: 970-491-6356
Email: france@cs.colostate.edu

ABSTRACT

The *Unified Modeling Language* (UML) is a standard modeling language in which some of the best object-oriented (OO) modeling experiences are embedded. In this paper we illustrate the role formal specification techniques can play in developing a precise semantics for the UML. We present a precise characterization of requirements-level (problem-oriented) Class Diagrams and outline how the characterization can be used to semantically analyze requirements Class Diagrams.
Keywords
Precise Semantics, Requirements Class Diagrams, UML, Z.

1 INTRODUCTION

The *Unified Modeling Language* (UML) [10] is a standard language for modeling complex systems from a variety of views using object-oriented (OO) concepts. The effectiveness of the UML as a standard is predicated, among other things, on there being a clear, precise, and pragmatic semantics for its notations. Informal semantics (i.e., semantics defined primarily in terms of key basic concepts that are not explicitly defined) can be ambiguous (if one is not aware of the intended implicit interpretations), can be incomplete and can contain inconsistencies. These problems can lead to confusion over appropriate use of the language and to the creation of models that do not clearly communicate their intent. Furthermore, subtle consequences that can lead to a deeper understanding of the concepts which, in turn, can result in more effective use of the language, can get lost in informal treatments of semantics.

Without a precise semantics, a standard modeling notation can devolve to a Tower of Babel, effectively diluting the utility of the standard. Formal specification techniques (FSTs) can play two significant roles in the development of a precise semantics for standard notations. They can be used as:

- tools for expressing formal semantics from which precise natural language descriptions can be obtained, and as
- tools that facilitate in-depth analyses of proposed interpretations.

In this paper a FST consists of a formal notation and mechanisms for rigorously analyzing statements expressed in the notation.

In the early stages of the development of a standard language informal discussions and debates on semantic issues, coupled with natural language statements reflecting particular perspectives, can yield valuable insights into poorly understood semantic concepts. There is a limit to the value of informal analyses of semantic concepts, and as the concepts become more varied and interdependent the need for more formal treatment of the semantics becomes apparent. The act of formalizing semantic concepts forces language developers to closely examine their perceptions of the concepts and to confront the assumptions underlying their understanding. Analyses of the resulting formal models can reinforce confidence in the interpretations, identify subtle deficiencies in the interpretations, or yield significant insights by revealing desirable or undesirable consequences of the interpretations.

Use of FSTs to explore semantic concepts paves the way for expressing the semantics formally. Having a mathematically-based definition of UML semantics is useful in that it provides a reference for resolving questions about meaning (or consequences of meaning) that cannot be directly answered by examining natural language descriptions. This is not to say that the semantics should be presented in the standard document using only formal languages. Bridging the gap between mathematical expressions and the real-world concepts they model requires much effort and can make communicating the semantics as problematic as using imprecise and vague natural language statements. A well-formulated natural language description of semantics that is derived from a formal expression of semantics is less likely to cause confusion than one that is not based on a formal semantics.

Producing mathematically-based formulations of UML semantics is *not* the primary goal of our work. Formal techniques are used in our work to gain insights and to explore the consequences of proposed semantics. The primary goal of our work is to develop precise semantics for UML notations, expressed in a form that is widely-understood (e.g., natural language), and that supports rigorous analyses of the models.

The intent of this paper is to demonstrate the role FSTs can play in the development of well-defined standard modeling languages. We illustrate the use of FSTs for exploring UML semantics by developing precise characterizations of the basic constructs of problem-oriented (requirements) Class Diagrams, and outline how the characterizations can be used to rigorously reason about the properties captured by the models. In section 2 we introduce the formal notation Z [11] that we use to explore the semantics of basic UML Class Diagram constructs. Z was used in our work because past experiences indicate that it provides adequate facilities for modeling OO concepts [6, 7, 4, 5]. In section 3 we give an overview of our formal characterization of basic Class Diagram constructs and discuss some of the issues raised by the characterization. In section 4 we illustrate how the characterization can be used to support rigorous semantic analyses of Class Diagrams and in section 5 we conclude with an overview of our planned work in this area.

2 THE Z NOTATION

In this section we introduce only the parts of the Z notation necessary to understand the specifications given in this paper (see [11] for more details).

The primary structuring construct in Z is the *schema*. A schema has two parts: a declaration and a predicate part. The *declaration* part consists of variable declarations of the form $w : Type$, where w is a variable name and $Type$ is a type name. The preceding declaration states that the value represented by w is a member of the set named by $Type$ (types are sets in Z). The *predicate* part consists of a predicate that defines the relationships among the declared variables.

Types in Z can be either *basic* or *composite*. Elements of basic types (or *given sets*) are used as the basic building blocks for more complex elements (elements with composite types). There are three kinds of composite types in Z: *set types* (elements are sets), *Cartesian product types* (elements are pairs of values), and *schema types* (elements are bindings of values to variables declared in a Z schema). The usual set operations (e.g., union, intersection) are available in Z. In particular, the number of elements in a finite set S is denoted $\#S$.

Using basic and composite types one can define relations as sets of pairs in which the first element is from the domain and the second element is from the range of the relation. For example, $R : A \leftrightarrow B$ is a Z declaration stating that R is a relation in which domain elements are drawn from A and range elements are drawn from B. A function in Z, for example $F : A \twoheadrightarrow B$, is a relation in which each domain element is related to exactly one range element. A sequence in Z is a finite partial function from natural numbers to the sequence element type. There are a number of operators defined for relations and functions in Z. In this paper we will use the operators described below. In what follows R is a relation declared as $R : A \leftrightarrow B$, and $R = \{(a1, b1), (a1, b2), (a2, b2), (a3, b1), (a3, b2)\}$.

- $\operatorname{dom} R$: Returns the set containing the elements in A that are related to elements in B. This set is called the *domain* of R. In this case $\operatorname{dom} R = \{a1, a2, a3\}$.
- $\operatorname{ran} R$: Returns the set containing the elements in B that are related to elements in A. This set is called the *range* of R. In this case $\operatorname{ran} R = \{b1, b2\}$
- $S \lhd R$ where S is a subset of A: Returns the subrelation of R that has a domain consisting of those elements of dom R that are also in S. For example, $\{a1, a2, a4\} \lhd R = \{(a1, b1), (a1, b2), (a2, b2)\}$. Note that the $a4$ element in S is ignored (because it is not in the domain of R).
- $R \rhd T$ where T is a subset of B: Returns the subrelation of R that has a range consisting of those elements of ran R that are also in T. For example, $R \rhd \{b1\} = \{(a1, b1), (a3, b1)\}$.
- $R(\!|\ \{S\}\ |\!)$ where S is a subset of A: Returns the set of all elements in the range that are mapped to elements in the set S in R. For example $R(\!|\ \{a2, a3\}\ |\!) = \{b1, b2\}$.
- $R^{\sim}$: Returns the inverse of R.

Z schemata are used to model both structural and dynamic properties of systems. A schema that captures the structural properties will be referred to as a *state schema*, and a schema that captures the dynamic properties will be referred to as an *operation schema*. In a state schema the components of a system are declared in the declaration section and the static properties of the system are defined in the predicate part. An example of a state schema is given below:

$[SUBS, TELEPHONES]$

TelService

$subs : \mathbb{P}\, SUBS$
set of service subscribers

$directory : SUBS \leftrightarrow TELEPHONES$
subscribers with telephones

$\mathrm{dom}\, directory \subseteq subs$
only subscribers can have telephones in this system

$\forall\, t : \mathrm{ran}\, directory \bullet \#(directory^{\sim} (\!|\, \{t\}\, |\!)) = 1$
a telephone can be associated with at most 1 subscriber

In the above, *SUBS* and *TELEPHONES* are Z basic types representing registered subscribers and registered telephones, respectively. *TelService* is a schema specifying the state of a telephone system consisting of a set of subscribers, *subs*, and a look-up facility, *directory*. The text in normal font that appears directly below a Z expression in the schema is a natural language description of the Z expression (i.e., comments). The predicate part of the schema contains two conjuncts (logical expressions written on separate lines in the predicate part of a schema are "anded" to form a single predicate).

An operation schema defines the relationship between the state at the start and at the end of an operation's execution. The declaration part of an operation schema declares variables representing the before and after states (as defined in a state schema), inputs, outputs, and any other variables needed to define the pre and postconditions. The predicate part of the schema defines the relationship between the before and after states. The following conventions are used for variable names in operation schemata:

unprimed variable (e.g., w) - value of variable before operation execution;
primed variable (e.g., w') - value of variable after operation execution;
variable ending in '?' (e.g., $w?$) - an input to the operation; and
variable ending in '!' (e.g., $w!$) - an output from the operation.
ΔS denotes a possible change in state defined by schema S (includes invariant properties of primed and unprimed state variables as defined in the schema S).
ΞS denotes that the state S does not change (includes invariant properties of primed and unprimed state variables as defined in the schema S and a predicate stating that the after state is identical to the before state).

An example of an operation schema is given below:

AddSub

$\Delta\, TelService$

$sub? : SUBS$
subscriber to be added

$sub? \notin subs$
input is not a current subscriber

$subs' = subs \cup \{sub?\}$
input is a subscriber at end of operation

$directory' = directory$
directory is unchanged by this operation

This operation schema defines an operation that adds a new subscriber to the telephone system whose state is defined by *TelService*. The predicate part consists of three conjuncts describing the relationship between the before and after states of the system determined by this operation.

3 A FORMAL CHARACTERIZATION OF REQUIREMENTS CLASS DIAGRAMS

The process that we used to explore the semantic foundation of static UML models can be described as follows:

Informal Analysis: A systematic and careful reading of the semantic and notation sections of the document was carried out in this phase. The objective was to gain an initial understanding of the concepts. This phase revealed some of the more obvious deficiencies in the UML document (e.g., the problem associated with the definition of association end multiplicity properties discussed later was identified at this stage).

Formalization: As stated above, informal analyses are inadequate when concepts become more varied and interdependent. More subtle deficiencies that can arise when concepts are intertwined to form more complex structures are more likely to be identified through rigorous analysis of the concepts. The formalization phase is concerned with expressing the concepts formally. This must be done in a manner that supports the analysis that will take place in the next phase. The formalization phase provides yet another opportunity for identifying deficiencies. For example, one can identify (1) omissions when there is not enough information to complete a formalization, (2) ambiguities when the information provided can be formally interpreted in different ways, and (3) inconsistencies when information results in an inconsistent formalization.

Rigorous Analysis: Rigorous analysis involves manipulating the formal expressions to derive consequences. Given the formalizations we explored their consequences by posing "what if" questions and manipulating the formal expressions to determine answers. We then tried to determine whether the answers were desirable or undesirable consequences of the meaning characterized by the formalization. For example, given a formalization of the *frozen* association property one can ask the question "What if the object at the frozen is deleted?" and determine whether the derived answer is desired or not by examining the OMG-UML document. We also identified desirable properties of UML constructs in the OMG-UML document that were not explicitly expressed in the formalization and attempted to prove that these were consequences of the formalization. If this could be done then this heightened our confidence in the formalization. If it could not be done then we examined the formalization to identify any problems in the expressions. This could happen if (1) an error was made in the formalization of the concept, (2) the concept being formalized was not well-understood, or (3) the relevant descriptions (the descriptions used to develop the formalization and the description of the property being checked) in the OMG-UML document are inconsistent.

A Class Diagram is a conceptual model of a system that is expressed in terms of classes and the relationships among them. At the requirements level a Class Diagram reflects a problem-oriented structure in which classes represent problem domain concepts. At the design level, a Class Diagram reflects a solution-oriented structure. The interpretation we use in this paper was developed for problem-oriented Class Diagrams. In the remainder of this paper a requirements-level Class Diagram is referred to as a CD. This paper outlines our formalization of CDs. A more detailed account of our formalization can be found in a technical report [3].

In our interpretation, a CD is a characterization of valid, externally observable system states. An externally observable *system state* is a structure consisting of all the system objects and links that can be observed by an external agent at some point in time. We refer to the system states that possess the properties expressed in a CD as *configurations*. The semantic domain for CDs is a collection of sets of configurations, and the meaning of a CD is a set of configurations.

The instance-based semantics of CDs we use is consistent with the UML object interpretation of a Class Diagram. The OMG UML document [10] states (pg. 2-61):

> The purpose of a model is to describe the possible states of a system and their behavior. The state of a system comprises objects, values, and links. ... The state of a system is a valid system instance if every instance in it is a direct instance of some element in the system model and if all of the constraints imposed by the model are satisfied by the instances.

References to "model" and "system model" in the above quote relate to CDs, and a "valid system instance" is a configuration in our interpretation.

CD constructs (e.g., classes and associations) may possess two types of properties: *static* and *dynamic*. Static properties are used to define the structure of elements represented by the construct, and dynamic properties are used to constrain how the elements are manipulated by operations. An example of a static property is the multiplicity of a class. If a class has a multiplicity, $m..p, m \leq p$, then in any valid state (configuration) there can be no less than m and no more than p objects of the class. An example of a dynamic property is the notion of *addonly* attributes. An addonly attribute is one that can hold more than one value, but once a value is added it cannot be removed. In our analysis, static properties of CD constructs are expressed in Z state schemata and dynamic properties are expressed in Z operation schemata.

In this section we focus on specifying the basic static and dynamic properties of general associations, compositions (strong aggregation), aggregation (weak aggregation), and generalization structures. For associations we consider only multiplicity and changeability properties in this paper. The navigability and visibility properties are considered not relevant to CDs (i.e., they reflect design considerations that should not appear in requirements models). Ordering and other secondary association properties are not considered in this paper.

3.1 Specifying Classes in Z

The set of all objects belonging to a class in a configuration is called the *configuration set* of the class, and the objects in the set are referred to as *configuration objects* of the class. In isolation, a class can be interpreted as the set of all its possible instances, called the *object space* of the class, but in the context of a CD a class defines its configuration sets (a subset of its object space). This interpretation of a class implies that constraints associated with a class in a CD are constraints on their configuration sets. For example, class multiplicity restricts the number of objects that can be in a configuration set of the class. This is consistent with the use of class constraints that we have encountered in

UML and other OO modeling notations. In Z, the object space of a class that is not a subclass is represented as a basic type.

In Z, the static properties of a class are defined by a state schema. For example, a class, *Cl*, with a multiplicity $m..p$, $(0 \leq m \leq p)$, attributes defined by another schema *CLattribs*, and with operations identified by elements of the type *OpIds* (a user-defined enumeration expressed as Z free types) is specified in Z as follows:

$[CL]$
Object space of Cl

Instance Schema

Cl

$cls : \mathbb{P}\, CL$
configuration set of Cl

$cl\_attribs : CL \rightarrow\!\!\!\!\!\!+ \; CLattribs$
maps objects to attribute values

$cl\_ops : CL \leftrightarrow OpIds$
maps objects to operation references

$m \leq \#cls \leq p$
multiplicity constraint

$\mathrm{dom}\ cl\_attribs = cls$

$\mathrm{dom}\ cl\_ops = cls$

The behavioral specification of an operation is given by a schema that maps operation references (elements of *OpIds*), the current state of class objects (attribute values and links), and parameter values (if any) to the set of all possible effects of the operation. An effect is defined as resultant state of the system (in our formalization, operations are viewed as atomic at the requirements level). We do not detail the formalization of attributes and operations in this paper. See [3] for a more in-depth discussion.

3.2 Formalizing binary associations

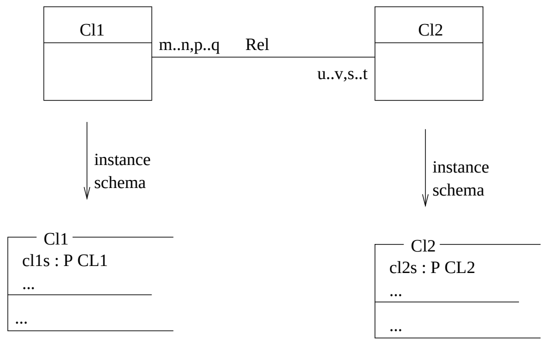
In a configuration, a binary association is interpreted as a mathematical relation between the configuration sets of the associated classes that satisfies the stated constraints on the association. A binary association is simply a set of object pairs, where each pair is called a *link*.

A careful reading of the UML document revealed that the terms "multiplicity", "range", "multiplicity specification" and "multiplicity property" are used loosely. The semantics part of the document does not mention "multiplicity specification" in the description of association semantics (page 2-57). One gets the impression from this section that an association end is associated with a single "multiplicity" which is defined as a "range of nonnegative integers" (use of the term range often implies a contiguous nature, this does not seem to be the case here). The notation part of the OMG-UML document gives a clearer picture of intent. It states that each association end has a multiplicity specification that defines a set of multiplicities ("a subset of the open set of nonnegative integers", page 3-68), and also gives examples of association-ends that are associated with multiple multiplicities.

Our understanding of multiplicity can be expressed as follows:

> An association-end has a multiplicity specification, where a multiplicity specification is a set of multiplicities, and a multiplicity is a nonnegative integer range.

Using the above we developed the following formalization of the multiplicity property. Consider the association *Rel* shown in the diagram below,



where $m..n$, $p..q$, $u..v$ and $s..t$ are ranges. The static properties of the association *Rel* are specified in Z as follows:

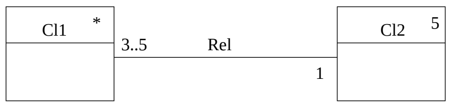
Rel

$Cl1$
$Cl2$
$rel : CL1 \leftrightarrow CL2$

$$\begin{aligned}
rel \in \{ & Rel : CL1 \leftrightarrow CL2 \mid \\
& \mathrm{dom}\ Rel \subseteq cl1s \\
& \wedge\ \mathrm{ran}\ Rel \subseteq cl2s \\
& \wedge\ (\forall x : cl2s \bullet m \leq \#(Rel^{\sim}(\!|\ \{x\}\ |\!)) \leq n \\
& \qquad \vee\ p \leq \#(Rel^{\sim}(\!|\ \{x\}\ |\!)) \leq q) \\
& \wedge\ (\forall x : cl1s \bullet s \leq \#(Rel(\!|\ \{x\}\ |\!)) \leq t\} \\
& \qquad \vee\ u \leq \#(Rel(\!|\ \{x\}\ |\!)) \leq v)\}
\end{aligned}$$

From the above formalization an interesting interplay between association-end multiplicity properties and a class' multiplicity property can be identified. Consider the case where m and p are greater than 0 and $m \leq p$. Then the expression:
$\forall x : cl2s \bullet m \leq \#(Rel^{\sim}(\!|\ \{x\}\ |\!)) \leq n$
$\vee\ p \leq \#(Rel^{\sim}(\!|\ \{x\}\ |\!)) \leq q$
implies that if there is at least one $CL2$ object in the system, then there must be at least m $CL1$ objects in

the system (or else the predicate is not true). If the multiplicity of the $CL2$ class is $x..y$, where $x > 0$, then the multiplicity of the $CL1$ class is constrained in that the lower bound must be greater than or equal to m. We are currently using the above formalization to identify categories of unnecessarily permissive association-end and class multiplicities. The characterizations of the categories that we develop can be used in CASE tools to identify multiplicity constraints that are too permissive. In the diagram below, the multiplicity of $CL1$ is too permissive: at any time there will be at least 15 instantiated $CL1$ objects in existence. A more appropriate multiplicity for $CL1$ is $15..*$.



3.2.1 Dynamic Properties of Associations

An association-end can be *changeable*, *frozen*, or *addonly*. A changeable association end is one in which no restrictions are placed on how links are set up between objects of the associated classes (the UML default).

If an association-end is frozen then the objects at the frozen association end are referred to as *target* objects, and those at the opposite end are referred to as *source* objects (e.g., see Fig. 1). The OMG-UML notion of frozen association ends is expressed as follows (pg. 3-67):

> The property {frozen} indicates that no links may be added, deleted, or moved from an object (toward the end with the adornment) after the object is created and initialized.

We interpreted this to mean that once a source object is created then no additional links can be added from the source object to target objects, and none of the links to target objects created during the creation of the source object can be deleted. A consequence of this is that if no links are created between the source object and target object when the source object is created then no links to target objects can be created during the lifetime of the source object.

In formalizing the UML description of frozen association ends we had to consider the question of what happens when an object at the frozen association end (a target object) is deleted. We could not find any answers in the OMG-UML document. Within our group most felt that an appropriate answer is that the link is also deleted. But others felt that this violated the constraint that no links can be removed from the source end *during the lifetime of the source object* (the emphasized text states their interpretation of the phrase "after the object is created and initialized" in the OMG-UML description of frozen properties). For example, consider

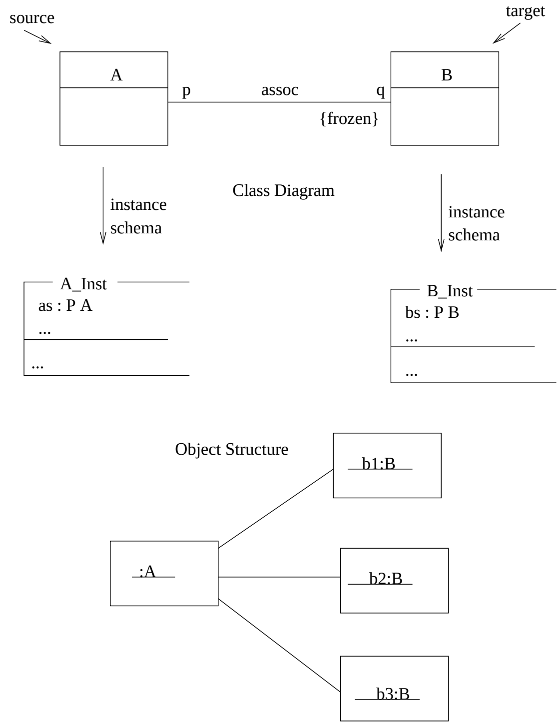


Figure 1: An example of a frozen association end

the object structure shown in Fig. 1, in which an A object is linked to three B objects ($b1, b2, b3$). The links between the A object and the B objects are frozen. It is clear that deletion of the A object would result in the destruction of the links (but not necessarily the target objects), but what happens if one of the B objects is deleted before its linked source object is deleted is not discussed in the UML document.

We could not find any additional information in the OMG-UML document to resolve this ambiguity so we defined two flavors of the frozen property: weak and strong frozen properties. If *assoc* is an association that has a strong frozen property at the association end connected to B (see Fig. 1), then the deletion of a B object in the object structure shown in Fig. 1 is *not allowed* until after the associated A object is deleted. This shade of frozen associations forces a death dependency between source and target objects: A linked target object can be deleted only after all its (frozen) linked source objects are destroyed (a target object can be linked to more than one source object if permitted by the association-end multiplicity specification). If *assoc* has a weak frozen property at the B end, then a linked B object can be deleted independently of its linked source object, resulting in the deletion of the corresponding link. In this case an *assoc* link is frozen as long as either linked object exists.

Let *Assoc* be a schema defining the static properties of the association *assoc* shown in Fig. 1:

$$\begin{array}{|l} \multicolumn{1}{l}{\_\_ Assoc \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\ A\_inst \\ B\_inst \\ assoc : A \leftrightarrow B \\ \hline \ldots \\ \hline \end{array}$$

The A configuration set, as, is defined in the schema A_inst and the B configuration set, bs, is defined in the schema B_inst. The two variants of the frozen property are expressed below:
Strong Frozen Property

$$\begin{array}{|l} \multicolumn{1}{l}{\_\_ FrozenDepAssoc \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\ \Delta Assoc \\ \hline as' \lhd assoc = \mathrm{dom}\, assoc \lhd assoc' \\ \hline \end{array}$$

Weak Frozen Property

$$\begin{array}{|l} \multicolumn{1}{l}{\_\_ FrozenIndAssoc \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\ \Delta Assoc \\ \hline (as' \lhd assoc) \rhd bs' = \mathrm{dom}\, assoc \lhd assoc' \\ \hline \end{array}$$

In the above schemata, $as' \lhd assoc$ is a domain restriction that returns the part of the *assoc* relation that involves only the domain elements in as' (the configuration objects in the after state). The symbol $\rhd$ represents range restriction and when applied to a relation R and a set of range elements S, as in $R \rhd S$, it returns the part of R that involves only the range elements that are in S (see section 2).

In the UML an association end is said to be *addonly* if links can be added to the source object, but none of the previously created links to the target objects can be deleted. Again, we define two shades of the addonly property which are formally characterized below for the association shown in Fig. 2:

$$\begin{array}{|l} \multicolumn{1}{l}{\_\_ StrongAdd \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\ \Delta Assoc \\ \hline \forall\, a : as;\ b : bs \mid (a, b) \in assoc \bullet \\ \qquad (a, b) \in assoc' \lor a \notin as' \\ \hline \end{array}$$

$$\begin{array}{|l} \multicolumn{1}{l}{\_\_ WeakAdd \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\ \Delta Assoc \\ \hline \forall\, a : as;\ b : bs \mid (a, b) \in assoc \bullet \\ \qquad (a, b) \in assoc' \lor a \notin as' \lor b \notin bs' \\ \hline \end{array}$$

In *StrongAdd*, once an *assoc* link is created between a and b it cannot be removed until the a element is destroyed. Consequently, the b element cannot be destroyed until after the a element is destroyed. In *WeakAdd* a linked b object can be deleted before its source a object is deleted (in which case the link is deleted).

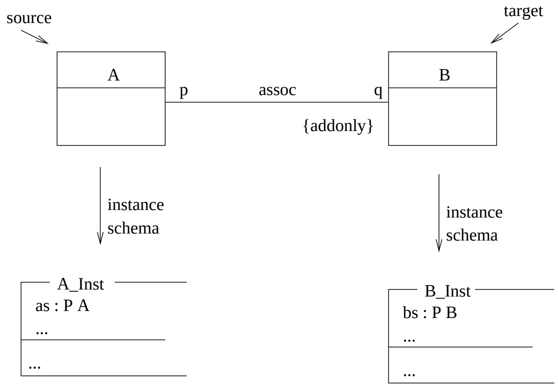


Figure 2: An example of an AddOnly association end

3.3 Formalizing aggregation

In the UML an aggregation corresponds intuitively to a whole/part association. Strong aggregation is called composition or a "composite aggregate" in the UML, and infers strong "ownership" of the parts by the whole. Weak aggregation infers weak "ownership". Distinguishing among (general) association, weak aggregation and composition requires formalizing of the UML notion of "ownership".

3.3.1 Formalizing simple composition and weak aggregation

The UML description of composition implies that the multiplicity specification at the whole end is a singleton set that consists of a single multiplicity that is either 1..1 or 0..1 (OMG-UML, pg. 2-57). A composition can be mathematically modeled as a function from the component configuration set to the whole configuration set. The form of the Z schema that captures static properties of a composition is the same as that for an association except that the composition relationship is modeled as a function. Structurally, the functional relationship between a part and its whole is a characteristic of a composition. Note that some associations that are not intended to be compositions can be modeled as functions, so this characteristic does not distinguish a composition from a general association.

The weak form of aggregation weakens the functional relationship between components and their aggregates to a general relation (allowing for the sharing of components). Weak aggregation between a component class and an aggregate class is thus structurally equivalent to a general association between the classes. One would

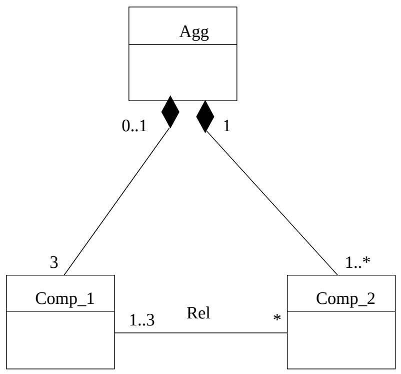


Figure 3: An encapsulating composition

then expect that the distinguishing features of weak and strong aggregation would appear in the form of dynamic properties. As we discuss later, our formalization of the dynamic properties of weak aggregation did not uncover any distinguishing features.

3.3.2 Formalizing encapsulating composition

In the UML a composition can "contain" associations (UML-OMG document, pg. 3-75).

> The meaning of an association in a composition is that any tuple of objects connected by a single link must all belong to the same container object.

We refer to a composition that "contains" associations as an encapsulating composition.

If *comp*1 and *comp*2 are two component classes of a whole class *whole* (where *comp*1*map* maps *comp*1 objects to their whole objects and *comp*2*map* maps *comp*2 objects to their whole objects), and *comp*1 and *comp*2 are related via an encapsulating association *rel*, then the property constraining the linking of *comp*1 and *comp*2 can be expressed as follows:

$$\forall\, r \in rel \bullet comp1map(first(r)) = comp2map(second(r)))$$

where $first(r)$ returns the first element in the pair r (a *comp*1 object) and $second(r)$ returns the second element in r (a *comp*2 object).

For example, the aggregation structure shown in Fig. 3 is formalized in the following schema (in what follows, the source of *rel*, defined in *Rel_Sc*, is *COMP_1* and the target is *COMP_2*):

```
┌─ AggConfig ─────────────────────────────
│ Agg_Inst
│ Instance Schema for Agg
│ Rel_Sc
│ Association Schema for Rel
│ comp1_agg : COMP_1 ↔ AGG
│ comp2_agg : COMP_2 ↔ AGG
├──────────────
│ ...
│ multiplicity properties
│ ∀ r ∈ rel • comp1_agg(first(r)) =
│             comp2_agg(second(r)))
└─────────────────────────────────────────
```

We also developed what we consider useful variations of encapsulating associations, by weakening and strengthening the OMG-UML property. These variations are discussed in [3]. A particularly useful stronger form of encapsulation that we developed requires that objects from classes involved in an encapsulating association *must* be linked if they occur as parts of an aggregate. This means that the parts must appear as pairs in an aggregate structure. For example, in a clinical laboratory system a test request can be modeled as a whole object consisting of test and sample object pairs (i.e., tests without samples, and vice versa, are not allowed in a test request).

3.3.3 Formalizing the dynamic aspects of aggregation

It is not clear to us what "coincident lifetimes" (OMG-UML, pg. 3-71) means in the OMG-UML document. A literal translation would result in the following interpretation: The parts are created at the same time the whole is created and they are destroyed when the whole is destroyed (consequently, parts are frozen for the lifetime of the whole). This interpretation contradicts the intent that a whole may "remove a part and give it to another composite object" (OMG-UML, pg. 2-57) and requires that all parts be associated with whole objects (i.e., disallowing multiplicities of 0..1 at the whole end).

We propose the following deletion property for compositions: If a whole in a composition is deleted then all the parts that are *currently* associated with the whole are deleted. This property allows for the removal of parts before the deletion of the whole (others have proposed similar properties, e.g., see [2, 9]).

Given a composition between a component *Comp* and a whole *Agg* that is defined by a schema *Comp_Agg*, the deletion property for compositions is specified as follows:

$$
\begin{array}{|l}
\hline
\textit{DelAgg} \\
\Delta Comp\_Agg \\
delaggs? : \mathbb{P}\, AGGS \\
\text{whole objects to be deleted} \\
\hline
delaggs? \subseteq aggs \\
\text{objects must be in start configuration} \\
aggs' = aggs \setminus delaggs? \\
\text{objects are deleted} \\
comps' = comps \setminus \\
\{c : comps;\ a : delaggs? \mid \\
\quad comp\_agg(c) = a \bullet c\} \\
\text{components are removed} \\
\hline
\end{array}
$$

In the previous section we noted that weak aggregation is structurally equivalent to general association. Unfortunately, the UML document does not provide enough information to make a distinction between the two concepts from either a structural or behavioral perspective. There are other ways of distinguishing weak aggregation from general binary association that we are currently exploring through formalization [8, 9].

3.4 Specifying Generalization Structures

The type space of a specialization hierarchy can be viewed as a carving up of the root superclass object space into subsets, where each subset is the object space of a subclass. In a configuration, the subclasses of a superclass are subsets of the superclass configuration set.

The static properties of a subclass, *Sub*, of a superclass characterized by an instance schema *Super*, that defines superclass configurations, *supers*, and for which the root superclass elements are drawn from the set $ROOT$, are defined in the following schema:

$$
\begin{array}{|l}
\hline
\textit{Sub} \\
Super \\
\text{inheritance of superclass properties} \\
subs, SUB : \mathbb{P}\, ROOT \\
\text{subclass config. set and object space} \\
\hline
subs = \{x : supers \mid x \in SUB\} \\
\hline
\end{array}
$$

The predicate states that the configuration set of the subclass are precisely those objects in the configuration set of *Super*, *supers*, that are in the object space of the subclass (SUB).

How a superclass configuration set is divided into subclass configuration sets can be constrained as follows:

- *Overlapping and Disjoint Subclasses*: A set of subclasses is said to be *disjoint* if there are no objects that are instances of more than one subclass in the set, otherwise the set is said to consist of *overlapping* subclasses. For subclass object spaces, $SUBS_1, \ldots, SUBS_n$ the disjoint property is stated as followed in Z: $\text{disjoint}\langle SUBS_1, \ldots, SUBS_n \rangle$. It is also possible to specify the disjoint property on subclass configuration sets (and allow object spaces to overlap). We suspect that the need for this type of constraint does not occur often. The OMG-UML document does not discuss whether the disjoint property applies to object spaces or configuration sets.
- *Abstract (Exhaustive) Superclasses*: An *abstract* superclass is one in which each superclass configuration object is also a configuration object of at least one depicted subclass. A superclass that can have configuration objects that are not configuration objects of any depicted subclass is said to be *non-abstract*. The property that a configuration set *supers* is abstract with respect to its subclass configurations is formalized in Z as follows: $\bigcup\{subs_1, \ldots, subs_n\} = supers$. This property can also be stated for superclass object spaces. Again, the OMG-UML document does not make it clear whether the property applies to object spaces or configurations.

A schema characterizing the static properties of a generalization hierarchy is created by including the schemata of the leaf subclasses in the declaration part (together these schemata define the static properties of all the classes in the hierarchy) and predicates expressing disjoint and abstract properties in the predicate part. An example of a formalization of a generalization structure will be given in the next section.

4 RIGOROUS SEMANTIC ANALYSIS

The precise characterization of a CD construct can be used to infer structural properties of constructs [1]. Using inference mechanisms (e.g., proof techniques) one can explore consequences of a particular interpretation.

The need to establish properties can also arise during application development out of a need to show that a model conforms to certain requirements, or out of challenges posed by reviewers of the models. Rigorous analysis may also be required to tackle questions for which answers are not explicitly given in the model.

Consider the top model in Fig. 4. This model involves a composition between the superclass and *SampleSlot*, and a *specialized composition* between a subclass and *SampleSlot*. A specialized association (composition) is one that involves a subclass, *Sub*, and another class, A, such that there exists an association with the same name between a class, *Super*, that is an ancestor of *Sub*, and the class A. The multiplicities on an association at the superclass level are constraints on the links that can be formed between objects of the superclass and objects of the associated class. Given that instances of subclasses are also instances of superclasses, the multiplicities at the superclass level also constrain the links that can exist between subclass objects and objects of the associated class. A modeler can further restrict the links at the subclass level by explicitly stating multiplicities on the association at the subclass level. These multiplicities must be consistent with the multiplicities given at the superclass level. This is the case for the specialized composition in the top model of Fig. 4.

In the top model of Fig. 4 one may ask whether an aggregation between *AdvanceAnalyzer* and *SampleSlot* is implied by the aggregation between *Analyzer* and *SampleSlot* and if so what are the constraints on the aggregation. An informal analysis of the CD leads to the conjecture that is expressed in UML terms in Fig. 4. The diagram expresses the conjecture that the aggrega-

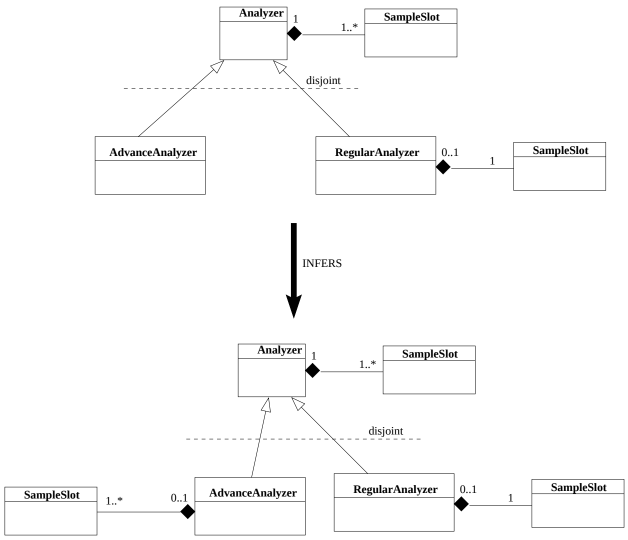


Figure 4: UML Inference Diagram

tion at the *Analyzer* level implies that an aggregation also holds between *AdvanceAnalyzer* and *SampleSlot*, where the multiplicity at the *SampleSlot* end is $1..*$ and the multiplicity at the *AdvanceAnalyzer* end is 0..1. The informal reasoning that produced this conjecture follows:

> An advanced analyzer is an analyzer, hence it can be associated with one or more sample slots. We note that this is true in the absence of further information that can further constrain the number of slots associated with an advanced analyzer. It is known that regular analyzers have sample slots and that regular analyzers are distinct from advanced analyzers (the respective classes are disjoint). From this we can conclude that a sample slot either belongs to a regular analyzer or to an advanced analyzer, but not to both at the same time. This implies that a sample slot is either associated with an advanced analyzer or it is not.

The specification characterizing the CD at the top of the "inference" diagram in Fig. 4 is given below:

$[ANALYZER, SLOT]$

Slot

$slots : \mathbb{P}\, SLOT$

Analyzer

$analyzers : \mathbb{P}\, ANALYZER$

AnalyzerConfig

Analyzer

Slot

$ADVS, REGS : \mathbb{P}\, ANALYZER$
object spaces for Advanced and
Regular Analyzers

$advs, regs : \mathbb{P}\, ANALYZER$
configuration sets for Advanced and
Regular Analyzers

$super\_agg, sub\_agg : SLOT ⇸ ANALYZER$
composition mappings

$advs = \{x : analyzers \mid x \in ADVS\}$
Advanced Analyzer config objects are precisely
those config objects of Analyzer
that are in the object space of Advanced Analyzer

$regs = \{x : analyzers \mid x \in REGS\}$
Regular Analyzer config objects are precisely
those config objects of Analyzer
that are in the object space of Regular Analyzer

$ADVS \cap REGS = \varnothing$
Subclasses are disjoint

$advs \cup regs = analyzers$
Superclass is abstract

$super\_agg \in slots \twoheadrightarrow analyzers$
Superclass composition

$sub\_agg \in slots ⤔ regs$
Subclass composition

$sub\_agg = super\_agg \rhd regs$
Subclass composition specializes superclass
composition

The conjecture that a sample slot is a part of zero or one advanced analyzer can be formally stated as follows:

$AnalyzerConfig \vdash \forall s : slots \bullet$
$((\exists a : advs \bullet super\_agg(s) = a)$
$\not\equiv (\forall a : advs \bullet super\_agg(s) \neq a))$ An outline of the proof of the above conjecture is given below:

- From the definition of *super_agg* we get:

  $$\forall s : slots; \exists an : analyzers \bullet super\_agg(s) = an$$

- Given that $analyzers = advs \cup regs$, the above is equivalent to: $\forall s : slots; \exists an : (advs \cup regs) \bullet super\_agg(s) = an$ which, in turn, is equivalent to: $\forall s : slots \bullet$
  $((\exists an : advs \bullet super\_agg(s) = an)$
  $\vee (\exists an : regs \bullet super\_agg(s) = an))$

- Given that $regs \cap advs = \varnothing$ the above implies: $\forall s : slots \bullet$
  $((\exists a : advs \bullet super\_agg(s) = a)$
  $\not\equiv (\forall a : advs \bullet super\_agg(s) \neq a))$ *End of Proof Sketch*

The conjecture that an advanced analyzer can have one or more sample slots (in the absence of information that can further constrain this property) can be expressed as follows:

$$AnalyzerConfig \vdash \forall a : advs \bullet \#(super\_agg^{\sim} (\!| \{a\} |\!)) \geq 1$$

The proof outline is similar to the above proof outline and so is not given in this paper.

A precise semantics can also lead to the development of transformation rules that can be employed within CASE tools to (1) transform complex diagrams to semantically equivalent simpler diagrams, and to (2) transform diagrams into semantically equivalent diagrams in which implicit properties are made explicit (the inverse is usually a simplification) . In [1] a small set of transformations are discussed. We have since enlarged this set with transformation rules for structures involving specialized associations. In the remaining part of this section we illustrate some of the rules we developed.

In defining the transformation rules for promoting and demoting specialized binary associations in generalization structures, a distinction is made between *complete* and *incomplete* CDs. A *complete CD* is one that states all the intended properties (i.e., the structure is not intended to be extended). An *incomplete CD* is one that does not state all the intended properties (i.e., the intent is that the structure will be extended). In general, one can be precise when reasoning about complete CDs. Constraints on possible extensions can be inferred from incomplete CDs as will be demonstrated in this section. For example, consider the CD in Fig. 5. If this CD is considered complete, then objects of D are linked only to objects of C. If the CD is incomplete, then there is the possibility that objects of D can be linked to objects of A that are not C objects.

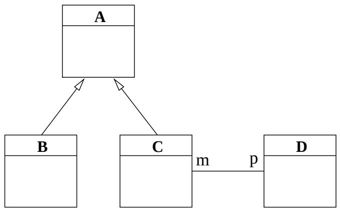


Figure 5: Complete vs. incomplete CD

The rules presented in this section have all been proven (in a manner similar to above). Space does not allow us to present proofs for the rules we give here. Instead we justify the rules informally. The informal presentation of the rules is based on insights gained while carrying out the proofs.

Demotion Rule 1: Association Demotion in a Complete CD

Fig. 6 gives the rule for the demotion of a binary association in a complete CD. In the left hand side (LHS) model, elements of class **A** are linked to elements of class **D**. Given that objects of a subclass are also objects of its superclass, then elements of the subclass can be linked to objects of class **D**. The CD is complete, meaning that the association between subclass objects and objects of the class **D** are constrained only by the superclass constraints. Given that a superclass object is associated with $p..q$ **D** objects, then each subclass object is also associated with $p..q$ **D** objects, as shown in the right hand side (RHS) diagram. In turn, a **D** object is associated with $m..n$ superclass objects. Some (including none) of the superclass objects can be **B** objects and some (including none) can be **C** objects. The number of **B** (**C**) objects associated with a **D** object can not exceed n.

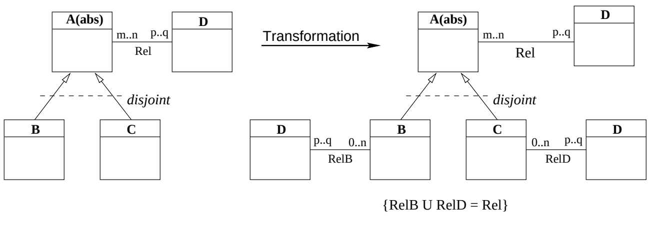


Figure 6: Demotion rule 1

Note that the inverse of this rule is a simplification: the LHS diagram is a simplification of the RHS diagram.

Demotion Rule 2: Association Demotion in an Incomplete CD

The LHS model in Fig. 7 is incomplete, meaning that additional information that can further constrain the association **Rel** when moved to the subclass level may be missing. In this case we cannot be precise about the cardinality at the subclass end of the association, but we can define the boundaries of the missing information pertaining to the association **Rel**. For example, at the **B** end we know that the cardinality has to be a subset of $0..n$. We also know that the union of the associations at the subclass level must be equal to the association at the superclass level. If this is not the case, then the properties expressed at the superclass level will be inconsistent with the properties expressed at the subclass level.

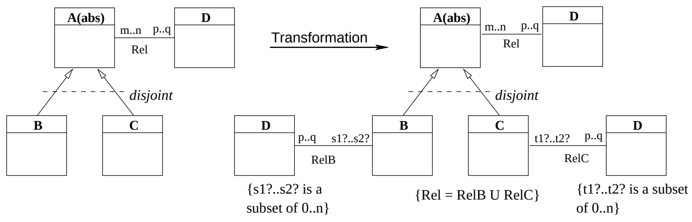


Figure 7: Demotion rule 2

We indicate an unknown value by placing a ? after the value reference. In the RHS model $s1?, s2?$ and $t1?, t2?$ are unknown cardinalities (completion of the model requires that these values be supplied). Constraints on unknown values are expressed as annotations enclosed in '{','}' in the CD.

Promotion Rule 1: Association Promotion in a Complete CD

The LHS CD in Fig. 8 is complete, meaning that the depicted association is only between objects of **C** and **D**. If this association is promoted, then only the **C** objects of **A**'s object space are mapped to **D** objects, the others are not. The assumption here is that the object space of **B** is not a subset of **C** (i.e. B is not a subclass of **C**). If the superclass is non-abstract this transformation still applies. The constraint on **RelA** in the RHS CD stipulates that objects of the superclass **A** are linked to either $p..q$ or zero **D** objects.

The inverse of this transformation is a simplification: the LHS diagram is a simplification of the RHS diagram.

Promotion Rule 2: Association Promotion in an Incomplete CD

The LHS CD in Fig. 9 is incomplete which means that the association shown can possibly involve other objects

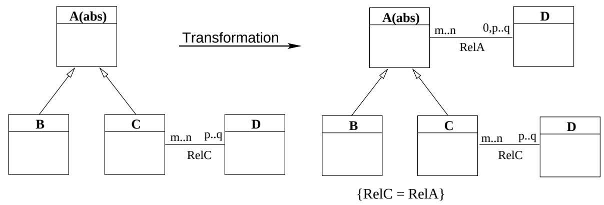


Figure 8: Promotion rule 1

of **A** that are not **C** objects. The cardinalities of the inferred association at the superclass level in the RHS model cannot be determined precisely, but we can infer that the cardinality at the **A** end must be a subset of $m..*$ (a **D** object is linked to at least m **C** objects). The cardinality at the **D** end is a subset of $0..*$.

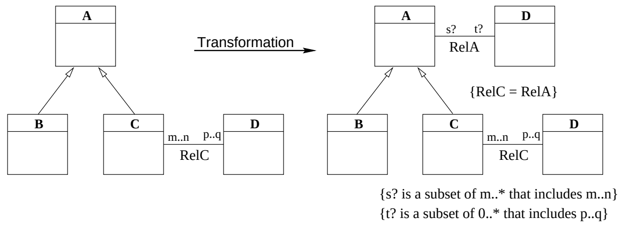


Figure 9: Promotion rule 2

5 CONCLUSION

In this paper we provide a glimpse of the role formal techniques can play in developing the UML. The primary goal of our work is to provide support for the UML that enables its use as a rigorous modeling language. This can be done by defining precise semantics for UML notations and developing mechanisms that allow developers to rigorously analyze the UML models. This work is being carried out as part of a collaborative effort to define a precise semantics for the UML that is undertaken by the *precise UML* (pUML) group. The approach taken by the pUML group is to use formal techniques to explore and define appropriate semantic foundations for OO concepts and UML notations, and to use the foundations to develop rules for transforming models to enable rigorous analysis. More information about the pUML effort can be found on the following website:
http://www.cs.york.ac.uk/puml/

We are currently working on expressing the Z formalizations of CDs in the *Object Constraint Language* (OCL) [10]. This exercise should provide us with insights into the ability of the OCL to formally express UML semantics.

In summary, we argue that mathematically-based techniques can and should be used to rigorously analyze the concepts upon which the semantics of a standard notation are based.

Acknowledgements

The work presented in this paper benefited from interactions with MIRG (Methods Integration Research Group) and the pUML group. In particular, I would like to thank Jean-Michel Bruel, Andy Evans, Brian Henderson-Sellers, and Bernhard Rumpe for the discussions we had around some of the issues presented in this paper. This paper is based on work supported by the National Science Foundation under Grant No. CCR-9803491.

References

[1] A. S. Evans. Reasoning with UML class diagrams. In *Workshop on Industrial Strength Formal Techniques (WIFT'98)*. IEEE Press, 1988.

[2] F.Civello. Roles for composite objects in object-oriented analysis and design. In *Proceedings of OOPSLA'93*, 1993.

[3] R. France, E. Grant, and J-M. Bruel. FUZed: An UML-based rigorous object-oriented analysis technique. CSU technical report, Colorado State University, Fort Collins, Colorado, USA, February 1999.

[4] Robert B. France, Jean-Michel Bruel, Maria M. Larrondo-Petrie, and Emanuel Grant. Rigorous object-oriented modeling: Integrating formal and informal notations. In *Proceedings of the 6th International AMAST Conference*, 1997.

[5] Robert B. France, Jean-Michel Bruel, and Gopal Raghavan. Towards rigorous analysis of fusion models: The mirg experience. In *Proceedings of the 2nd Northern Formal Methods Workshop, Springer-Verlag*, 1997.

[6] J. Anthony Hall. Using Z as a specification calculus for object-oriented systems. In D. Bjørner, C. A. R. Hoare, and H. Langmaack, editors, *VDM and Z – Formal Methods in Software Development*, volume 428 of *Lecture Notes in Computer Science*, pages 290–318. VDM-Europe, Springer-Verlag, New York, 1990.

[7] J. Anthony Hall. Specifying and Interpreting Class Hierarchies in Z. In Jonathan P. Bowen and J. Anthony Hall, editors, *Z User Workshop, Cambridge 1994*, Workshops in Computing, pages 120–138. Springer-Verlag, New York, 1994.

[8] H. Kilov and J. Ross. *Information Modeling: An Object-Oriented Approach*. Object-Oriented Series. Prentice Hall, 1994.

[9] R. B. France M. Saksena and M. M. Larrondo-Petrie. A characterization of aggregation. In *5th. International Conference on Object Oriented Information Systems*. Springer, 1998.

[10] The Object Management Group (OMG). Unified Modeling Language. Version 1.3, OMG, http://www.omg.org, June 1999.

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