Applying the Theory-Model Paradigm

David Alex Lamb
Andrew Malton
Xiaobing Zhang

February, 1996
External Technical Report
ISSN-0836-0227-
1996-401

Department of Computing and Information Science
Queen's University
Kingston, Ontario, Canada K7L 3N6

Version 1.1
Document prepared May 9, 1997

Contents

List of Figures

List of Tables

1 Introduction

This is a tutorial on applying the Theory-Model paradigm to the specification of software engineering analysis and design methods. Its primary audience is students who need to learn to write such descriptions. We expect the primary readers of this report will be students doing Theory-Model descriptions, and students in Queen's software engineering courses who wish to understand the basics of the descriptions that we supply them in the course.

1.1 What is the Theory-Model Paradigm?

The *Theory-Model Paradigm* is a way of describing and understanding design methods, designs, and design verification. The ideas originate in the "model theory" of mathematics. There, a *theory* is a collection of expressions or statements made in an formal language of uninterpreted operations and symbols. For example, *group theory*, having expressions over variables, one constant, one unary operation, and one binary operation, is a collection of equations induced by the usual rules of equation and a certain set of five axioms. An *interpretation* of a theory gives meaning to the uninterpreted symbols of the theory. For example, an interpretation of group theory might involve letting variables vary over integers, letting the constant be 0, the unary operation be negation, and the binary operation be addition. A *model* of the theory is an interpretation that satisfies all the axioms. Thus, the integers under addition are a model of group theory. A different interpretation, might not be a model of the theory: you may not let the binary operation be multiplication over the integers.

We apply these ideas to software design in the following way:

- A design method corresponds to a theory: it introduces certain categories; its design rules and their consequences correspond to theorems about the categories.

- A design, that is the work produced by a designer, corresponds to an interpretation: a particular set for each category.

- A design is "correct" if it follows the design rules; correspondingly, an interpretation is valid (i.e., is a model of the corresponding theory) if it satisfies all the axioms.

Design *verification* corresponds to testing whether the design is a model of the corresponding theory. Since, in general, the sets and design rules we are dealing with are all finite, we know that it is theoretically possible to verify designs, by searching for contradictions to the axioms. We hope it will also prove feasible.

The development of a design theory in enough detail to use in design verification proceeds as follows:

1. Develop a formal description of the design method, using mathematical notation where possible for precision. Lately, we have been using the Z notation for this step. It is the hardest step of the three, requiring about 80% of the effort in developing a validation capability.

2. Develop a data model for the theory. Many of the checks required for design verification are tedious to do by hand. We thus expect in the long term to have automated support for design verification, although in the short term we often work on small examples by hand. A tool for design validation must be able to represent the design information, and so it is appropriate to develop a data model for it using typical data modeling methods. At present we use entity-relationship diagrams for this purpose. There is usually a close correspondence between some of the sets described in the mathematical theory and some of the entity sets of the entity relationship model, but the relationship is not necessarily direct.

3. Develop an executable form for the checks of whether any axioms are violated. In the past we have used Prolog for this purpose; in the future we are considering using Goedel.

There are feedback loops among the steps; while developing the data model, for example, one might find a simpler method of expressing a concept from the formal description.

Design verification is not the only use for design theory formalizations. The act of formalization itself is useful, since it forces us to make precise many aspects of the material that are imprecise in the original sources. By comparing the formalizations of similar design methods, we can draw exact and detailed comparisons among the methods.

The present report focuses on the first step: preparing a precise mathematical description of the original source material.

1.2 Basic Rhetorical Principles

The Theory-Model paradigm is fundamentally intended to give precise but readable descriptions of design methods. This means that concerns of rhetoric (the art of conveying meaning) are at least as important as the associated technical concerns.

For precision, we use the Z notation[Spiv92] to write mathematical descriptions. There are several reasons for this.

- Z has a well-defined syntax and semantics, to which we can refer people who wish to understand the notation that we use. This mitigates the problem of readers having to learn an idiosyncratic notation for writing mathematics.

- Available with Z is a type-checker, `fuzz`, which can help catch many simple mistakes in drafts of the mathematical expressions.

- Z provides a structuring mechanism, the "schema", which we can use to organize our specifications.

A fundamental rhetorical principle for this work is that the formal description cannot stand on its own. We formalize design methods whose original authors have already described them informally. Between the original informal description and our formal description there must be additional material, in informal but technically precise language. This is called the *technical description*.

The correct view to take is that the technical description should be able to stand on its own, with the formal description to make it absolutely precise. In principle it should be possible to obtain a sensible (though possibly imprecise) view of the design theory by omitting the formal description and leaving only the technical. Beware of the temptation, therefore, of focusing so intently on recording a formal description that the technical description lapses. This is an easy trap to fall into after becoming familiar with the formal notation. While many styles of composition are appropriate, in our view it is best to develop the technical and formal descriptions concurrently. The technical description must be written eventually, and each helps make the other clearer as choices are taken and revised.

The concurrent refinement of the technical and formal description proceeds until the point at which the former, while perfectly clear, uses concepts with no direct formal translation. The formal description at this point is an *encoding* of the technical description. At this point, formalization normally stops.

The situation somewhat resembles that of a software requirements document and the code that implements it. The requirements correspond to the informal description; the technical description corresponds to the software design document and code comments; the formal description corresponds to the code. Well written design documents and comments can stand alone. Software engineers generally accept that they should write design documents and comments with the implementation.

Since your purpose in formalizing is precise communication, expect to follow the normal rules of style for any essay or technical paper. In particular, a formalization requires an introduction setting a context, and each new section of the formalization should have prose describing how that section fits into the whole.

1.3 Original Materials

The first step of formalization is to develop a reasonable understanding of the original source material. This requires at least one thorough reading of the original sources, and possibly more. Formalization often requires decisions about how to represent things precisely which the original material defined imprecisely. In documenting such decisions, you should cite the original sources carefully, supplying page numbers or page ranges for the discussion of the imprecise concepts.

Beware of the temptation to improve the original source in some way. Every lack of precision in the original is an occasion of this temptation, because you must make some choice about how to fill in missing details. Of course, this is a great benefit of formalization: forcing you to be precise. However, the value of a formalization is lessened wherever it diverges from the original. If you make real improvements to the *method*, as opposed to improvements to the *degree of precision*, you should identify them as such, and document them separately.

2 The Formalization Task

In this section (the main body of this paper), we explain a process for developing and recording design theories. We use the formalization of Rumbaugh's Object Modeling Technique (OMT) as our main example[Rumb91], primarily because it is the material we were working on when the need for this paper arose.

2.1 Design Categories - Introducing Types

You must decide early what are the basic categories of the design theory. In our recent work, we have been formalizing software design methods in which elements of the design have both graphical and textual descriptions. During the formalization, we must develop representations for each class of graphical elements. Some elements appear to be able to exist on their own; others can exist only in the context of some more fundamental element. For example, in Rumbaugh's OMT, "classes" and "associations" can exist on their own, but "attributes" only exist with their class or attribute.

For each basic category of the formalization, we introduce a type to correspond to it. The formalization language, Z, is strongly typed: new types can be introduced as needed. We choose one of three ways of introducing a type to formalize a category.

- Introducing a "given set".
- Introducing a subset of some previously introduced set.

- Introducing a “schema”, the variables of which formalize essential properties of design elements in that category.

On recognizing a basic category, we believe the right formalization to try first is the introduction of a new given set. For example, in formalizing OMT you might introduce given sets as follows.

$$[\mathit{CLASS}, \mathit{ASSOCIATION}]$$

2.1.1 Combining Given Sets

Since Z is strongly typed, two different given sets have no elements or operations in common. As you progress with the formalization, however, you sometimes discover that two basic categories, previously identified as distinct, share similarities. Strong typing prevents you from sharing assertions and operators between these two categories. You are led instead to three alternatives:

- Duplicate Z text for both types. This violates the “separation of concerns” principle.
- Withdraw the disjoint given sets, replacing them with a single given set, to represent their union. You then define the assertions and operators once only, and introduce the two originally separate categories as subsets of the union set. Unfortunately, this reduces the amount of checking available with `fuzz`, since it only checks types, and different subsets of the same given set have the same type.
- Combine the two given sets in a disjoint union, which can be clumsy and hard to read.

To retain as much type checking as possible, and to avoid duplicate assertions, we have chosen the third alternative.

In OMT, classes and associations share common features, and so we prefer to consider them together as *entities*, and introduce the disjoint union:

$$\begin{aligned}\mathit{ENTITY} ::= {} & \mathit{ClassToEntity}\langle\!\langle \mathit{CLASS} \rangle\!\rangle \\ \mid {} & \mathit{AssociationToEntity}\langle\!\langle \mathit{ASSOCIATION} \rangle\!\rangle\end{aligned}$$

This declares a new give set, ENTITY, and two functions, $\mathit{ClassToEntity}$ and $\mathit{AssociationToEntity}$, which map classes and associations, respectively, to entities. Each of these functions is a *total injection*; that is, each CLASS element maps to a distinct ENTITY element. Moreover, the elements corresponding to classes and those corresponding to associations *partition* the set of entities: each entity corresponds to either a class or an association.

In OMT, “roles”, “operations”, and “attributes” appear at first glance to be significantly different. It would seem natural to introduce three given sets. But they have this in common:

- Whenever they appear they must be related to their class or association (that is, to their "entity").

- Any of them can be named, although roles are not necessary named. Within an entity scope, the names of any role, operation, or attribute must be unique.

- For classes we will eventually need to represent "inheritance", by which subclasses inherit attributes and operations from superclasses. This suggests that we will eventually be writing a collection of predicates about inheritance, which will apply equally to attributes and to operations.

The similarities lead us to consider at least that attributes and operations should be formalized as subsets of some more general set; and we'll include roles as well. Roles, attributes, and operations will be known collectively as *features.*

$$[ATTRIBUTE, ROLE, OPERATION]$$

$$\begin{aligned} FEATURE ::= {} & AttributeToFeature\langle\langle ATTRIBUTE\rangle\rangle \\ \mid {} & RoleToFeature\langle\langle ROLE\rangle\rangle \\ \mid {} & OperationToFeature\langle\langle OPERATION\rangle\rangle \end{aligned}$$

2.1.2 Schemas for Design Categories

Schemas (see below) arise in Z specifications for collecting together several pieces of related information. Formally, a Z schema means a subset of a cross product; it is an "indexed product" because the individual components of the product have individual names in Z (which are called *variables*). When elements of a design category are characterized by certain essentials, it might well be appropriate to formalize the category as a schema, and so collect the essential characteristics together as components of the product. There are, however, three phenomena that prevent or condition using a schema to formalize a basic category:

- a requirement for *unique identity*,

- a lack of *uniformity*, and

- the need for an *even* presentation style.

Since a schema means an indexed product, instances of a schema type are distinguished only by the values of their component variables. However, design methods often allow for two elements of a basic category to be present in a design with identical attributes. Their *unique identity* is not characterized by their attributes but by context. For example, in OMT one may introduce two different classes with exactly the same attributes and exactly the same operations. The only difference, in OMT, is their name. Sometimes this can be dealt with by adding "name" as an attribute, but this is really not appropriate. In general, the scope rules for naming (see below) are dependent on context, not on the identity of the thing named. If names were attributes, local scope would be difficult to discuss; comparing different designs be impossible, if the different designs used the same name for different classes.

Since a schema means an indexed product, every attribute (variable) in the product must always be present. Optional attributes cannot be introduced as schema variables, and if a basic category lacks the *uniformity* of a small set of required attributes, formalizing it as a schema will probably be difficult to understand.

A basic principle of rhetoric is to introduce new information *evenly*, without overwhelming the reader with too much at once. In our formalizations we take care to minimize the amount of new information the reader must absorb in any one place. Especially in mathematical writing, the average reader requires a "rest" after absorbing a chunk of symbolic information. We prefer therefore to introduce new design categories as small schemas, formalizing the essentials of the design elements as a small collection of schema variables.[1] Subordinate design properties and relationships are formalized later, by introducing new schemas focussed on the subordinate material.

2.1.3 Infinite Types but Finite Sets

In any software design method there are basic categories; a particular design is made up of elements of those categories, related to one another according to the design rules of the method. A design category, viewed as a set of possible design elements, is usually *infinite*, because there are infinitely many *possible* designs. The elements of a particular design, viewed as a subset of possible design elements, is always *finite*, because any given design is a finite construction. We use given sets, as above, to formalize the infinite potential of a basic category, and below we shall introduce schematic sets to represent their *extents*: their finite subsets that are the elements of particular designs. This style, of introducing infinite types first, and finite subsets later, is a frequently occurring pattern in the Theory-Model paradigm as we practice it.

2.2 Introducing Schemas

We usually use "schemas" to state the required relationships between design categories. Typically the relationships of interest to us in a particular design

[1]In Section 5.1 we discuss an alternative approach advocated by Ryman. He considers that schemas interfere with a clean presentation of design theories.

apply not to *all possible* design elements of the category, but only to those elements which have actually introduced into our present design. This aspect may be called the *design-time formalization* of a theory. We introduce schema variables to stand for finite subsets of the design categories, and additional data describes the properties of the particular design. It is often the first schema we introduce, and can be called the "basic" schema of the design-time formalization.

Let's apply these ideas to deciding on a representation of the design-time categories of attributes, operations, and roles in OMT. We have already introduced our basic categories, but a particular design will use a finite set of particular instances of them. The notions of "entity" and "feature" will reappear as finite subsets of the general notions introduced above. They are the information content of a particular design.

In our OMT formalization, *ENTITY* is the type of potential design elements that are classes or associations; a given design will involve particular classes and associations, formalized as a set variable *entity* in the basic schema, having subsets *class* and *association*.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{OMTentity}} \\
\hline
\mathit{entity} : \mathbb{P}\, \mathit{ENTITY} \\
\mathit{class} : \mathbb{P}\, \mathit{CLASS} \\
\mathit{association} : \mathbb{P}\, \mathit{ASSOCIATION} \\
\mathit{classEntity}, \mathit{associationEntity} : \mathbb{P}\, \mathit{ENTITY} \\
\hline
\mathit{classEntity} = \mathit{ClassToEntity}(\!|\mathit{class}|\!) \\
\mathit{associationEntity} = \mathit{AssociationToEntity}(\!|\mathit{association}|\!) \\
\mathit{entity} = \mathit{classEntity} \cup \mathit{associationEntity} \\
\hline
\end{array}
$$

We need five sets:

- The three sets of entities, classes, and associations in a design
- The two subsets of entities corresponding to classes and associations.

FEATURE is the type of potential design elements which are roles, attributes, or operations; we treat it similarly to entities:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{OMTfeature}} \\
\hline
feature : \mathbb{P}\, FEATURE \\
role : \mathbb{P}\, ROLE \\
attribute : \mathbb{P}\, ATTRIBUTE \\
operation : \mathbb{P}\, OPERATION \\
roleFeature, attributeFeature, operationFeature : \mathbb{P}\, FEATURE \\
\hline
roleFeature = RoleToFeature(\!| role |\!) \\
attributeFeature = AttributeToFeature(\!| attribute |\!) \\
operationFeature = OperationToFeature(\!| operation |\!) \\
\\
feature = roleFeature \cup attributeFeature \cup operationFeature \\
\hline
\end{array}$$

To collect the rules about basic concepts for future reference, we define a schema that combines the properties of the previous schemas.

$$OMTBasic \mathrel{\widehat{=}} OMTentity \wedge OMTfeature$$

This construct, called *schema conjunction*, essentially means that *OMTBasic* includes all the definitions and predicates of both *OMTentity* and *OMTfeature*.

2.3 Relations and Functions

If essential properties of design elements are formalized by schema variables; optional or inessential properties are modelled by functions and relations.

In OMT and similar design methods, features never exist independently, but are always associated with some entity. Given a particular feature in a valid design, we can discover the particular class or association in which it was introduced. This is even true in the presence of inheritance, since there is always an original point of introduction, in some ancestor class, for inherited attributes and operations. In mathematical English we would say something like “for every feature, there is a corresponding point of definition in some class or association”. Such a “for every ... there is a ...” suggests a formalization as a function in the mathematical sense. Here, the function *definedIn* has type $FEATURE \nrightarrow ENTITY$, because it yields an *ENTITY* in the design (the introduction point) for any *FEATURE* element introduced in the design. It is a partial function ($\nrightarrow$) instead of a total one ($\longrightarrow$) because we do not wish to insist that each *FEATURE* necessarily have a corresponding *ENTITY*; we wish to make few claims about the set of all possible features and entities, restricting ourselves to their properties in the context of particular object models.

Sometimes the relationships called for in a design theory don't suggest a functional style of formalization, but a more general relational style. However, it is best to write a relation in a functional style if possible. A functional style lends itself to equations, and equational presentations and reasoning are clearer and more elegant. It is possible, however, to use an equational style with relations; and it is possible to write elegant Z without often using equations. In this tutorial we have tried to use an equational style where possible.

Sometimes, perhaps often, you want to formalize using a relation that is the inverse of a function: if $f \in A \mathrel{+\!\!\!\!\rightarrow} B$ then $f^{\sim} \in B \leftrightarrow A$. In OMT, one speaks sometimes of "all the features of a class" or "all the features of a particular association". One is speaking of a set (of features) which is a function of a class; this can be viewed as a general relation between classes and features, which happens to be the relational inverse of the function *definedIn* alluded to above. Alternatively it can be viewed directly as a set function of a class; in Z, the relational image notation can be used to refer to the set: "all the features of class c" is $\mathit{definedIn}^{\sim}\llparenthesis\{c\}\rrparenthesis$.

We have sometimes approached formalization in the past with a "minimalist" attitude, in which we attempted to define a single expression of each idea. However, since our purpose in specification is to communicate clearly and precisely, the aim is not minimality but readability. It is usually helpful to introduce by name all the relations and functions that we will be using regularly later. If we will be writing equations in each direction of a relation, we give names to each direction. We have mentioned *definedIn* formalizing the relationship between each feature and its defined class; we may also need *featureOf* formalizing the inverse relation.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{OMTFeature}} \\
\hline
\mathit{OMTBasic} \\
\mathit{definedIn} : \mathit{FEATURE} \mathrel{+\!\!\!\!\rightarrow} \mathit{ENTITY} \\
\mathit{featureOf} : \mathit{ENTITY} \leftrightarrow \mathit{FEATURE} \\
\hline
\mathit{definedIn} \in \mathit{feature} \longrightarrow \mathit{entity} \\
\mathit{featureOf} \in \mathit{entity} \leftrightarrow \mathit{feature} \\
\mathit{featureOf} = \mathit{definedIn}^{\sim} \\
\hline
\end{array}
$$

A recommended style is to follow each schema with a list of natural language explanations, one per predicate. Thus:

- Every *feature* has a corresponding *entity*; *definedIn* is a (total) function, according to the $\longrightarrow$ arrow.
- There may be more than one *feature* for each *entity*; *featureOf* is a general relation, according to the $\leftrightarrow$ arrow.
- *featureOf* is the relational inverse of *definedIn*.

The beginner finds the repeated structure of declaration and axiom peculiar. Why should we have to introduce the general type above the line in the "declaration part", and then say it again below the line in the "predicate part"? It is because Z does not use dependent types. Identifiers *introduced* in the declaration part can only be *used* in the predicate part. Consequently, because '*entity*' and '*feature*' are declared in *OMTBasic*, they cannot be used in declarations in the schema above. We are reduced to stating the requirements we really want only in the predicate part, where any declared identifier can be used.

2.4 Names and Naming

In this business it is important not to confuse levels of abstraction. We always find ourselves using Z terms where we should be using technical terms from the formalization, or from the original theory. In particular, Z identifiers are completely different from "names" as they may be used in the design theory. If we introduce $example : CLASS$ in a schema, then *example* is the Z identifier of some member of the set *CLASS*; in no way is this to be confused with some "name" `example` which might be the name of a class in a particular design. That would be confusing domains of discourse.

The purpose of *CLASS* and *ENTITY* and so forth is to distinguish and clarify our knowledge of these categories of the design theory. Different members of these sets, with their unique identity, correspond to different design elements. Elements such as class diagrams and associations (represented by lines or diamonds in a diagram) have a unique identity irrespective of any name that they have in the design. With a CASE tool, their unique identity may be reflected in the positioning of some collection of boxes or lines in a particular place on the screen. Their names may be reflected as strings of characters near the boxes.

When formalizing names, you may perhaps choose strings, but unless you need to model naming conventions explicitly, it is unnecessary to distinguish names to this level of detail. Instead, simply introduce a given set of names.

$[NAME]$

In formalizing names in a design theory you must pay attention to scope rules that the source material establishes and follows. Typically all names in some context must be unique: in independent or parallel context names can be reused. For example, in OMT a class may not have two attributes with the same name; but two different classes could each have completely unrelated attributes with the same name. The context is called a *scope*. A *scope rule* establishes the scopes, and then legislates that within a scope, names and named things must be in one-to-one correspondence. This suggests two sets: the names used in the scope, and the things named. It also suggests two functions: the name of a thing (a function of its identity), and reference to a thing (a function of names).

In OMT, the naming of entities is simpler than the naming of features, because entities have global scope. A *named entity* is an entity with a name. An *entity name* is a name of a named entity. Entity names and named entities are in one-to-one correspondence (a bijection, given by the $\rightarrowtail\mkern-18mu\twoheadrightarrow$ arrow).

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{OMTEntityNamesBasic}} \\
\hline
\textit{OMTBasic} \\
\textit{namedEntity} : \mathbb{P}\, \textit{ENTITY} \\
\textit{nameOfEntity} : \textit{ENTITY} \mathrel{+\mkern-12mu\rightarrow} \textit{NAME} \\
\textit{entityName} : \mathbb{P}\, \textit{NAME} \\
\textit{theEntity} : \textit{NAME} \mathrel{+\mkern-12mu\rightarrow} \textit{ENTITY} \\
\hline
\textit{namedEntity} \subseteq \textit{entity} \\
\textit{namedEntity} = \operatorname{dom} \textit{nameOfEntity} \\
\textit{entityName} = \operatorname{ran} \textit{nameOfEntity} \\
\textit{nameOfEntity} \in \textit{namedEntity} \rightarrowtail\mkern-18mu\twoheadrightarrow \textit{entityName} \\
\textit{theEntity} = \textit{nameOfEntity}^{\sim} \\
\hline
\end{array}
$$

Note how the sets and functions are introduced in a general way as subsets and functions over the design categories; and then the precise requirements are stated as predicates. They are partial functions of the design categories, because a given design only uses finitely many of the potential design elements of the method; they are bijections of the design-time categories, for reasons explained above.

We have not finished with OMT entity naming, though. In OMT, classes must have names, but associations needn't. There is no way in Z to state that a property is not required; one may only be explicit about such matters in the technical description.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{OMTEntityNames}} \\
\hline
\textit{OMTEntityNamesBasic} \\
\hline
\textit{classEntity} \subseteq \textit{namedEntity} \\
\hline
\end{array}
$$

The naming of features is slightly more complex, because the scope of feature names is not global. The scope for features is the entity with which they're associated. We formalize local scope by the same method as the global scope above, except that each of the sets and functions is parameterized by the scope itself. The naming operations are thereby given a context, which is the scope.

Within a given entity (class or association), then, a *named feature* is a feature with a name. A *feature name* is a name of a named feature. Feature names and named features are in one-to-one correspondence.

$$
\begin{array}{|l}
\hline
\mathit{OMTFeatureNamesBasic} \\
\mathit{OMTFeature} \\
\mathit{namedFeature} : \mathit{ENTITY} \longrightarrow \mathbb{P}\ \mathit{FEATURE} \\
\mathit{nameOfFeature} : \mathit{ENTITY} \longrightarrow (\mathit{FEATURE} \mathrel{+\!\!\!\!\rightarrow} \mathit{NAME}) \\
\mathit{featureName} : \mathit{ENTITY} \longrightarrow \mathbb{P}\ \mathit{NAME} \\
\mathit{theFeature} : \mathit{ENTITY} \longrightarrow (\mathit{NAME} \mathrel{+\!\!\!\!\rightarrow} \mathit{FEATURE}) \\
\hline
\forall\, e : \mathit{entity} \bullet \\
\qquad \mathit{namedFeature}(e) \subseteq \mathit{featureOf}\,\llparenthesis\{e\}\rrparenthesis \wedge \\
\qquad \mathit{namedFeature}(e) = \mathrm{dom}(\mathit{nameOfFeature}(e)) \wedge \\
\qquad \mathit{featureName}(e) = \mathrm{ran}(\mathit{nameOfFeature}(e)) \wedge \\
\qquad \mathit{nameOfFeature}(e) \in \mathit{namedFeature}(e) \rightarrowtail\!\!\!\!\twoheadrightarrow \mathit{featureName}(e) \wedge \\
\qquad \mathit{theFeature}(e) = (\mathit{nameOfFeature}(e))^{\sim} \\
\hline
\end{array}
$$

Both attributes and operations must be named; roles need not be. Again we can formalize the constraint, but not the license. Since attributes, operations, and roles partition features,

$$
\begin{array}{|l}
\hline
\mathit{OMTFeatureNames} \\
\mathit{OMTFeatureNamesBasic} \\
\hline
\forall\, e : \mathit{entity} \bullet \\
\qquad \forall\, f : \mathit{attributeFeature} \cup \mathit{operationFeature} \mid e \mapsto f \in \mathit{featureOf} \bullet \\
\qquad\qquad f \in \mathit{namedFeature}(e) \\
\hline
\end{array}
$$

To collect the rules about names for future reference, we define a schema that combines the properties of the previous names-related schemas.

$$\mathit{OMTNames} \mathrel{\widehat=} \mathit{OMTFeatureNames} \wedge \mathit{OMTEntityNames}$$

2.5 Formalizing Properties of Design Elements

Sets introduced for design categories have members which formalize design elements. This captures the identity of design elements. However, design theories usually allow for additional properties of design elements, optional for their category; and impose or allow for further relationships between elements.

For example, in OMT a “role” of an “association” may or may not have a name but it must have at least two other data:

- the class whose “objects” can play the role in an association
- the “multiplicity”, which governs how many objects can play the role (in a dynamic sense).

In general, this kind of relationship-determining data can be formalized in either of two ways.

- Represent the data collectively as variables of a new schema which formalizes the relationship.
- Represent each datum separately, as we did with names, as a function of the design category.

The first alternative seemed attractive to us at first, since it gathered all the information about a type in one place. However, as Section 3.5 describes, we eventually decided that it leads to unnecessary complications. Thus we find the second alternative to be generally better.

Every role has a multiplicity and an associated class; thus we introduce

$$
\begin{array}{|l}
\hline
\textit{OMTRoleProperty} \\
\textit{OMTBasic} \\
\textit{player} : \textit{ROLE} \mathrel{+\!\!\!\!\to} \textit{CLASS} \\
\textit{multiplicity} : \textit{ROLE} \leftrightarrow \mathbb{N} \\
\hline
\textit{player} \in \textit{role} \longrightarrow \textit{class} \\
\operatorname{dom} \textit{multiplicity} = \textit{role} \\
\hline
\end{array}
$$

The definition of *player* is similar to ones we have seen before; every role in a design has a single class that plays that role. *multiplicity* is slightly more subtle.

Multiplicity represents the number of instances of classes that can play the given role in the given relation. In standard databases one thinks of simply having a range of cardinalities, which can be represented by pair of numbers. In OMT the restrictions are allowed to be more general, thus requiring a set of natural numbers for the representation.

- A one-to-one relationship would be represented by a multiplicity being the set consisting of single number 1.
- An optional role would be represented by the set consisting of the numbers 0 and 1.
- A "many" side of a relation would be represented by a multiplicity consisting of all the natural numbers.

This suggests that *multiplicity* might be considered a function from roles to sets of natural numbers. However, for reasons discussed later, we prefer to replace functions returning sets by general relations.

We also introduce the following schema to show the relationship between the basic OMT categories and roles:

OMTRole

$$\begin{array}{|l}
\mathit{OMTRoleProperty} \\
\mathit{isRoleOf} : \mathit{ROLE} \nrightarrow \mathit{ASSOCIATION} \\
\mathit{roleOf} : \mathit{ASSOCIATION} \leftrightarrow \mathit{ROLE} \\
\hline
\mathit{isRoleOf} \in \mathit{role} \longrightarrow \mathit{association} \\
\mathit{roleOf} = \mathit{isRoleOf}^{\sim}
\end{array}$$

2.6 Evolving a Formalization

Our main application of the Theory-Model paradigm is formalzing software design methods, which are typically defined in large books. It is common in such books to cover basic features first, then introduce advanced features separately. This will sometimes mean that decisions you make earlier in the formalization process require revision (or, at least, reconsideration) later.

What we have for roles is adequate for most kinds of associations one finds in an object model in OMT. However, two of the advanced features complicate our representation. In both generalization and aggregation, there can be several sets that play a particular role. In aggregation there is a single container class and possibly several "part of" classes. In generalization there is a single superclass and potentially several subclasses.

2.6.1 Aggregation

In the earlier stage of our thinking about this problem it seemed necessary to introduce several different roles to represent subclasses of a generalization and similarly several different roles to represent the different parts in an aggregation. The formalization began to look more and more complicated as we introduced several new schemas and other elements to represent properties of generalization and aggregation. We considered introducing special elements to distinguish the parts in a fixed aggregation, special representations for the subclasses and superclass in a generalization, and so on.

Whenever things look like they are getting too complicated, it is wise to look over the material again to see if one can discover an appropriate simplification. For example, we considered attempting to represent all the different parts of a fixed aggregate as one role, where the role simply had several distinct players. This would have required changing the *OMTRoleProperty* schema so that the *player* function (from *ROLE* to *CLASS*) became a general relation.[2] This introduces its own complications, but would let us talk about "the part-of role" as meaning all the parts of an aggregation. Unfortunately, Rumbaugh's book gives examples of aggregates where, for example, a microcomputer is an

[2]This is our preferred approach, instead of having it become a function returning a set of classes.

aggregate of a monitor, a system box, a mouse, and a keyboard, and each of those parts of the microcomputer aggregate could potentially have different multiplicities. For example, there are one or more monitors and zero or one mice[3]. Thus, we are forced to regard an aggregate as having many different "part of" roles. This vindicates our original idea that the "part of" elements of an aggregate could each be thought of as a distinct role – but it required considerable examination of the original material to ensure that this was an essential aspect of the formalization, and not just a convenience.

The fundamental question arises at this point: can we still view generalizations and aggregations as associations, or must we introduce a new category? If aggregates were to be completely different from other kinds of associations, they would require a new Z base type (i.e. a new given set) as their representation in the formalization. However, aggregates still seem to have roles, just as simple associations do, but require additional information and additional structure in their descriptions.

Thus, we introduce a given set to represent aggregates, and a schema to represent the basic information associated with an aggregation:

$$[AGGREGATE]$$

$$\begin{array}{|l}
\hline
\textit{OMTAggregateBasic} \\
\textit{OMTRole} \\
\textit{aggregate} : \mathbb{P}\, \textit{AGGREGATE} \\
\textit{collects} : \textit{AGGREGATE} \leftrightarrow \textit{ASSOCIATION} \\
\textit{collectedIn} : \textit{ASSOCIATION} \mathrel{+\!\!\!\!\to} \textit{AGGREGATE} \\
\hline
\textit{collectedIn} \in \textit{association} \mathrel{+\!\!\!\!\to} \textit{aggregate} \\
\textit{collects} = \textit{collectedIn}^{\sim} \\
\hline
\end{array}$$

We define the set of aggregates in a particular design (*aggregate*), and the relationship between associations and the aggregates that collect them.

Each aggregate can be considered as a container that contains one or more parts. We represent this information as a pair of relations:

$$\begin{array}{|l}
\hline
\textit{OMTAggregateProperty} \\
\textit{OMTAggregateBasic} \\
\textit{containerOf} : \textit{AGGREGATE} \leftrightarrow \textit{ROLE} \\
\textit{partOf} : \textit{AGGREGATE} \leftrightarrow \textit{ROLE} \\
\hline
\operatorname{dom} \textit{containerOf} = \textit{aggregate} \\
\operatorname{ran} \textit{containerOf} \subset \textit{role} \\
\\
\operatorname{dom} \textit{partOf} = \textit{aggregate} \\
\operatorname{ran} \textit{partOf} \subset \textit{role} \\
\hline
\end{array}$$

[3]Figure 3.22 on page 38 of Rumbaugh's book.

An aggregate isn't just an arbitrary collection of associations. We need to express several consistency requirements:

$$
\begin{array}{|l}
\hline
\textit{OMTAggregate} \\
\textit{OMTAggregateProperty} \\
\textit{containerClass} : \textit{AGGREGATE} \pfun \textit{CLASS} \\
\hline
\textit{containerClass} \in \textit{aggregate} \longrightarrow \textit{class} \\[1ex]
\forall\, \textit{ag} : \textit{aggregate} \bullet \forall\, a : \textit{collects} \llparenthesis \{ \textit{ag} \} \rrparenthesis \bullet \\
\qquad \#(\textit{roleOf} \llparenthesis \{ a \} \rrparenthesis) = 2 \\[1ex]
\forall\, a : \textit{aggregate} \bullet (\textit{containerOf} \fatsemi \textit{player}) \llparenthesis \{ a \} \rrparenthesis \\
\qquad = \{ \textit{containerClass}(a) \} \\[1ex]
\forall\, a : \textit{aggregate} \bullet \forall\, r : \textit{containerOf} \llparenthesis \{ a \} \rrparenthesis \bullet \\
\qquad \textit{multiplicity} \llparenthesis \{ r \} \rrparenthesis = \{ 1 \} \\[1ex]
\forall\, a : \textit{aggregate} \bullet \\
\qquad \textit{containerOf} \llparenthesis \{ a \} \rrparenthesis \cup \textit{partOf} \llparenthesis \{ a \} \rrparenthesis = \\
\qquad \textit{roleOf} \llparenthesis \textit{collects} \llparenthesis \{ a \} \rrparenthesis \rrparenthesis \\
\hline
\end{array}
$$

- Each aggregate has a single "container" class.
- All associations collected in each aggregate must be binary (that is, have exactly two roles).
- The containers of an aggregate must all have the same class as their player.
- Every part will have exactly one container, so all container roles have multiplicity 1.
- The containers and parts of an aggregate must be all the roles of the associations making up the aggregate.

2.6.2 Generalization

The visual representation of generalizations in Rumbaugh's method seems superficially similar to that for aggregation. We spent considerable thought on whether they were similar enough to have a common formalization. At first, we thought that the various classes used as super or subclasses in an generalization do not really represent distinct "roles" since the concept of multiplicity, which applies to roles, did not seem to apply to subclasses or superclasses.

Eventually, Ryman pointed out that we could view a generalization as like an aggregation of injection functions, in just the way that we had injection functions from *CLASS* and *ASSOCIATION* to *ENTITY* in our formalization

(Section 2.1.1 on page 5). In addition to the usual properties of aggregates, a generalization would have an additional restriction on the multiplicity of the subclass roles.

Generalizations appear to have new property, distinct from those of other associations: that of a "discriminator". Discriminators would seem to be a form of a label for a generalization, specifying the basis of the generalization. Thus, Rumbaugh gives the example of class Vehicle, which could be discriminated by its propulsion mechanism, and also by its operating environment. On page 39 Rumbaugh says that the discriminator is simply a name for the basis of the generalization. In the examples we see that many generalizations have no discriminator, and Rumbaugh says that the discriminators are optional. Thus, discriminators would appear to be closely resembling names of associations, and at this point in our formalization there does not seem to be a need to introduce a separate category for discriminators.

Rumbaugh requires one additional property of generalizations: the diagrams have either an open triangle, where the subclasses are disjoint, or a closed triangle, where they overlap. Thus for example "musician" might have overlapping subclasses for "pianist" and "cellist". To represent this, we would introduce variables *disjoint* and *overlapping*, both being sets of generalizations. Unfortunately, Z has no Boolean data type, so we can represent a Boolean property only via enumerating the set of elements that have that property.

2.6.3 Revisions

In the previous section we did not present a formalization of generalization, since it would be similar to that for aggregation. In fact the two are sufficiently similar that we should consider merging them, as *CLASS* and *ASSOCIATION* are each *ENTITIES*. Thus *AGGREGATE* and *GENERALIZATION* might each be kinds of *GROUPING*; *GROUPING* in turn might be another kind of *ENTITY*, or a kind of *ASSOCIATION*.[4]

Regardless of the specific choice we make, incorporating the new pieces of the formalization requires going back to edit old ones. We regard such revision cycles as normal, perhaps even necessary; it is quite similar to what one must do with any expository prose.

2.7 Putting the Pieces Together

We early wrote of needing to separate out different elements of the formalization, to avoid overwhelming the reader. One must eventually put all the parts together to form an entire formalization. We do so in defining a new schema

[4]These considerations are very similar to those for designing a class hierarchy for an object-oriented system.

which incorporates the properties of all the previous schemas, thus OMT is the conjunction of the component schemas:

$$\begin{array}{l} \mathit{OMT} \mathrel{\widehat{=}} \mathit{OMTBasic} \land \mathit{OMTFeature} \land \mathit{OMTNames} \\ \quad \land \mathit{OMTRole} \land \mathit{OMTAggregate} \end{array}$$

3 Things People Did Wrong

In experimenting with aspects of Z, we have tried several idioms that did not work out well for us, or which we eventually decided were the wrong ways to do things. In reading beginners' specifications, we have found similar problems. This section summarizes the mistakes, and what we think writers should do instead.

3.1 Distinct Names

There is a natural tendency to try to phrase predicates similarly to the original natural language. We originally phrased the "distinct names" predicate as "distinct entities have distinct names;" phrased more formally, this became "given two entities, different from each other, their names are different," which became the following predicate:

$$\begin{array}{|l} \hline \mathit{xOMTDistinctEntityNames} \\ \mathit{OMTBasic} \\ \mathit{nameOfEntity} : \mathit{ENTITY} \pfun \mathit{NAME} \\ \hline \mathit{nameOfEntity} \in \mathit{entity} \pfun \mathit{NAME} \\ \forall\, e1, e2 : \operatorname{dom} \mathit{nameOfEntity} \mid e1 \neq e2 \bullet \\ \qquad \mathit{nameOfEntity}(e1) \neq \mathit{nameOfEntity}(e2) \\ \hline \end{array}$$

The requirement can be phrased more elegantly as "there is a one-to-one correspondence (a bijection) between named entities and their names," as we do on page 12.

We expect that some readers might prefer avoiding a "complicated" concept like bijection, believing the above formulation to be simpler and easier to understand. However, we prefer to avoid quantifiers over elements of sets when we can express the same concept easily using operations on the sets themselves. This might be viewed as another instance of the classic dichotomy between using a smaller and "simpler" vocabulary (and consequently writing longer sentences) versus a larger "more technical" vocabulary (and consequently writing smaller sentences). Both positions have some merit, but we choose to regard bijection as such a fundamental concept that it is well worth using when it is applicable.

3.2 Quantifying Over Set Expressions

People seem reluctant to quantify over set expressions. For example, the first predicate of *OMTAggregate* on page 17 includes the quantifier

$$\forall\, a : \mathit{collects} \llparenthesis \{ \mathit{ag} \} \rrparenthesis \bullet \ldots$$

We have seen many initial attempts at such quantifiers that were phrased in the style

$$\forall\, a : \mathit{association} \mid a \in \mathit{collects} \llparenthesis \{ \mathit{ag} \} \rrparenthesis \bullet \ldots$$

which is unnecessarily longer and more complex.

3.3 Repeated Predicates

In an earlier version of the *OMTFeatureNames* schema on page 13, we wrote the predicates this way:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{XOMTFeatureNames}} \\
\mathit{OMTFeatureNamesBasic} \\
\hline
\forall\, e : \mathit{entity} \bullet \\
\qquad (\forall\, a : \mathit{attributeFeature} \mid e \mapsto a \in \mathit{featureOf} \bullet a \in \mathit{namedFeature}(e)) \land \\
\qquad (\forall\, o : \mathit{operationFeature} \mid e \mapsto o \in \mathit{featureOf} \bullet o \in \mathit{namedFeature}(e)) \\
\hline
\end{array}$$

The parallel structure of the two clauses violates the fundamental principle that one should state a common property in only once place. Our present formalization combines the two into a single quantifier of the form

$$\forall f : \mathit{attributeFeature} \cup \mathit{operationFeature} \ldots$$

3.4 Eliminating Quantifiers

Often a predicate written using elements of sets can be simpler if written using the sets themselves; this may simplify the predicates by eliminating quantifiers and the corresponding quantifier variables. For example,

$$\forall\, x : S \bullet x \in T$$

is simpler as

$$S \subset T$$

Z provides many useful operations for treating sets as a whole, instead of element-by-element; Table 1 shows several of them.

$R1 \fatsemi R2$	relational composition	$\{ r1 : \mathrm{dom}\, R1;\ r2 : \mathrm{ran}\, R2 \mid (\exists\, y : \mathrm{ran}\, R1 \bullet r1 \mapsto y \in R1 \land y \mapsto r2 \in R2) \bullet (r1, r2) \}$
$S \lhd R$	domain restriction	$\{s : S;\ r : \mathrm{ran}\, R \mid s \mapsto r \in R\}$
$R(\!\vert S \vert\!)$	relational image	$\{s : S;\ r : \mathrm{ran}\, R \mid s \mapsto r \in R \bullet r\}$

Table 1: Set Operations

3.5 Functions versus Schemas

In Section 2.5, we introduced function *player* and relation *multiplicity* to represent properties of roles. At one time we believed it was better to collect all such properties into a schema, such as:

```
┌─ Role ─────────────────────────────
│ role : ROLE
│ player : CLASS
│ multiplicity : ℙ₁ ℕ
└────────────────────────────────────
```

Some predicates needed to quantify over members of the *role* set; to talk about the corresponding properties, we then needed a way to find the schema representing the properties of a role, given the role itself:

```
┌─ xOMTRole ─────────────────────────
│ OMTBasic
│ roleData : ROLE ⇸ Role
├──────────
│ dom roleData = role
│ {r : role • (roleData(r)).player} ⊂ class
│ ∀ r : role • (roleData(r)).role = r
└────────────────────────────────────
```

We abandoned this scheme in favour of the present one (representing properties as separate functions) for several reasons:

- The need to use the *roleData* function complicated several of our predicates, conflicting with our basic rhetorical principles.
- Using a schema in this way requires that *every* element of the design category in question possesses the properties to be described. Names, for example, cannot be formalized as variables of a schema describing roles in OMT, because not all roles have names.
- Defining a schema for a design category requires that we discuss all the properties of the design category in one place. It makes it more difficult for us to distribute our discussion of properties among separate sections of the formalization.

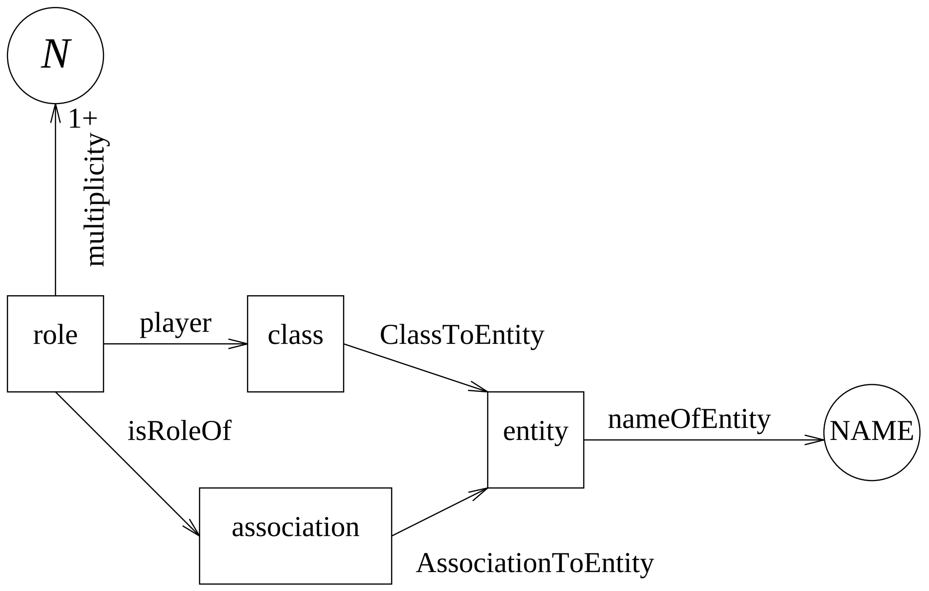

Figure 1: Example Entity-Relationship Diagram

4 Data Model and Execution

Section 2 focussed on the formalization task, since it is the most difficult one for most members of our target audience. The Theory-Model approach also includes developing a data model of the formalization, and an executable representation of the axioms. We discuss these steps in this section.

4.1 Data Model

The second phase of the Theory-Model approach is to develop a data model of the material formalized in the first phase. We use a variant of the Entity-Relationship approach[Chen88] for this.

Translation of our style of Z specification to an E-R model is fairly straightforward, aside from the rhetorical issue of how large a diagram to make and which items to group in each diagram. Most given sets become entity sets, represented by boxes; predefined types, such as $\mathbb{N}$, become value sets, represented by ovals. Functions and relations become relationship sets, usually represented by arrows, which point in the direction implied by the name of the Z function or relation.

Figure 1 shows an E-R diagram of some of the concepts of roles and entities. In some E-R conventions there would be a special notation for the injection functions *ClassToEntity* and *AssociationToEntity*, which can be viewed as defining a generalization. Most E-R conventions of which we are aware forbid any cardinality except 1-1 for “attributes” such as the *multiplicity* function.

In some early applications of the Theory-Model paradigm[Lamb90], we presented the collection of E-R diagrams in a section of the document separate from the formal description. We now believe this to be a mistake; regardless of the order in which they are developed, the diagrams and the formal prose complement each other, and should be presented together.

4.2 Executable Representation

Our primary aim for the formal prose and information model is clear communication with other people about the meaning of the original work we are describing. An executable version of the axioms is unnecessary for this purpose. However, we believe a formalization effort should still proceed to develop an executable representation, as this checks the soundness of the formalization.

We have developed a reasonably straightforward method for translating our style of formal description into Prolog, which falls into four main steps:

- Represent the entity sets.
- Represent the relationship sets
- Represent basic integrity constraints, such as type and multiplicity constraints.
- Represent other axioms of the formalization.

Each member x of a given set M is represented by a Prolog ground fact, of the form $M(x)$, asserting that its single argument is an element of the set, and defining x's identity. There are several possibilities for representing entities:

- A simple method is to pick Prolog atoms formed by appending a number to a base name. Thus, for example, roles might be represented as

  ```
  role(role_001).
  role(role_002).
  ```

- If an entity set has a collection of "attributes" sufficient to uniquely identify its elements, you might use the representation of the attribute values as the unique identities. Thus, for example, each class must have a unique name:

  ```
  class("employee").
  class("department").
  ```

- If a collection of given sets form a single generalization hierarchy, you might consider using the representation of the parent of the hierarchy for the children as well. Thus, for example, classes and associations are both entities:

```
class(entity_001).
class(entity_002).
association(entity_003).
```

With this approach, it would be natural to represent the parent by a rule rather than by enumeration:

```
entity(X) :- class(X).
entity(X) :- association(X).
```

You can represent n-ary relationship sets by n-ary Prolog predicates. Thus we might represent part of the information about an "employee works-in department" relationship as:

```
nameOfEntity(entity_001,"employee").
nameOfEntity(entity_002,"department").
nameOfEntity(entity_003,"works in").
player(role_001,entity_001).
isRoleOf(role_001,entity_003).
```

Integrity constraints are straightforward. In general, constraints are verified by searching for a counter-example; on failure, the constraint is considered to be satisfied.

- Type constraints on relationships can be checked by rules that search for elements of the wrong type:

  ```
  badNameOfEntityFirstArg(X) :-
    nameOfEntity(X,_),
    not(entity(X)).
  ```

- Cardinality constraints on functions and relations can be checked by stylized rules that build sets and check their sizes.

More complex axioms, such as those for aggregate consistency in Section 2.6.1 on page 17, may require hand-translation.

4.3 Iterative Development

We have presented the three activities of the Theory-Model paradigm (formalization, diagramming, and executable representation) as though they were sequential steps. Instead, the "later" stages can feed back to the "earlier." If a concept proves difficult to diagram directly, you may need to revise both the diagram and the original formal prose. If an axiom proves difficult to translate into an executable form, and seems important enough to be worth checking, you may need to revise the formal description.

We have found that some people prefer to begin immediately with the entity-relationship diagrams, and only later move on to the formal prose. We see no harm in this, if all three forms eventually exist.

5 Conclusion

We have shown how to apply the Theory-Model paradigm and basic principles of rhetoric to writing readable design theories (formalizations of software design methods).

As yet there exist no complete formalizations in this style; we have applied a less well-developed set of stylistic guidelines to Shlaer-Mellor Object-Oriented Analysis[Zhan94], and are working on one for Rumbaugh's OMT.

5.1 Difficulties with Z

A few properties of Z got in the way of our rhetorical principles.

In Section 2.3, we pointed out that Z inherently seems to require a "double definition" of most concepts because of its scope rules. It would seem possible to define an extension to Z allowing for names introduced in a definition part to be reused elsewhere in the definition, if there is some topological sort of the definitions that avoids circularity.

We use schemas as a modularity mechanism, breaking up a large specification into units. Were this our only use of schemas, it would be possible to argue that they buy us nothing that isn't already conveyed by the table of contents of the paper. If we made all our declarations global, the "double definition" problem would vanish, and later definitions could make use of earlier ones. We have been reluctant to take this step, but in private conversations some of our colleagues have suggested that it might be appropriate when clarity of communication is paramount.

5.2 Acknowledgements

The Theory-Model paradigm was proposed by Ryman [Ryma89], and applied in a preliminary form to the development of the ImagEdit product of IBM. From Ryman we have learned many of the principles of writing readable Z descriptions.

Zhang and Lamb learned Z from a short course by McMorran[McMo93], where he emphasized several rhetorical principles for readability.

References

[Chen88] P. Chen, "The Entity-Relationship Model - Towards a Unified View of Data," in J. Mylopoulos, editor, *Readings on Artificial Intelligence and Databases*, pages 98-111, Morgan Kaufmann Inc. (1988).

[Lamb90] David Alex Lamb, Nitin Jain, and Arthur Ryman, "A Theory-Model Formalization of Jackson System Development," Technical Report ISSN-0836-0227-90-269, Queen's University Department of Computing and Information Science (October 1990). Published simultaneously as IBM TR-74.051.

[McMo93] Mike McMorran and Steve Powell, *Z Guide for Beginners*. Blackwell Scientific Publishers (1993).

[Rumb91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen, *Object-Oriented Modeling and Design*. Prentice-Hall, Inc. (1991).

[Ryma89] Arthur G. Ryman, "The Theory-Model Paradigm in Software Design," Technical Report TR 74.048, IBM Corporation (October 1989).

[Spiv92] J. M. Spivey, *The Z Notation: A Reference Manual*. Prentice-Hall, Inc. (1992).

[Zhan94] Xiaobing Zhang, "A Theory-Model Formalization of Shlaer-Mellor Object-Oriented Analysis," in *Proceedings of IBM Centre for Advanced Studies Conference (CASCON'94)*, pages 324-333, Toronto, Ontario (1994).