Exploring The Semantics of UML Type Structures with Z

R. B. France, J.-M. Bruel, M. M. Larrondo-Petrie, and M. Shroff*
Department of Computer Science & Engineering
Florida Atlantic University
Boca Raton, FL-33431, USA.
Email: {robert,maria}@cse.fau.edu

* *Laboratoire IRIT/SIERA*
F-31062 Toulouse Cedex, France

Abstract

The Unified Modeling Language (UML) builds upon some of the best object-oriented (OO) modeling concepts available, and is intended to serve as a common OO modeling notation. Given its intended role, it is important that the UML notation have a well-defined semantic base. In this paper we present some early results from our work on the systematic formalization of UML modeling constructs. The paper focuses on the formalization of UML Class Diagrams. The formal notation Z is used to express the semantics of Class Diagrams.

Keywords

Formal Specification Techniques, Object-Oriented Analysis and Modeling, Unified Modeling Language, Z.

1 INTRODUCTION

The *Unified Modeling Language* (UML) (Booch *et al.*, 1997) is a proposed common object-oriented (OO) modeling notation, currently being developed

© IFIP 1997. Published by Chapman & Hall

by some of the more experienced OO methodologists. The potential primary strengths of UML constructs, their simplicity and intuitive appeal, are also potential sources of problems. A significant problem is UML's reliance on informally defined semantics. This can lead to situations where models are interpreted differently because of differing viewpoints on what the semantics are. This is more likely to occur when complex structures (e.g., those involving recursive structures) are involved.

A formal semantic base for the notation allows one to rigorously reason about the models being built. Our work on formalizing other OO and structured notations indicates that the ability to rigorously analyze models strengthens validation and verification of the models. In our past work we have used formalized semantic bases for graphical techniques to animate requirements models, and to statically analyze properties (e.g., see (Bruel *et al.*, 1996; France *et al.*, 1997)).

In this paper we present a Z (Spivey, 1992) formalization of the UML constructs used to build Class Diagrams consisting only of types and their associations. Such diagrams can be used to model the static structure of systems at the requirements level. We assume that the reader is familiar with the Z notation. In section 2 we give the current form of our rules for transforming Class Diagram constructs to Z, and in section 3 we show how they can be used to formalize a non-trivial Class Diagram. We conclude in section 4 with an overview of our future work on the formalization of the UML notation.

2 FORMALIZING UML ANALYSIS-LEVEL CLASS DIAGRAMS

A Class Diagram is a model of the static structure of a system expressed in terms of classes, types, objects (class instances) and their associations. A UML *type* is a specification of concrete UML classes. In UML, classes implement types, that is, classes provide concrete implementations of the attributes and operations abstractly defined in types. We will refer to Class Diagrams that consist solely of types as *Type Diagrams*. Type Diagrams provide appropriate abstractions for modeling problems at the requirements analysis phase of software development, and is the focus of the formalization given in this paper.

In our formalization, a Type Diagram characterizes a set of instance structures, referred to as *valid instance structures* or *configurations*. A configuration is one that exhibits the properties expressed in the Type Diagram. One can view a configuration as a snapshot of a system's structure at some point in time, where the instances are those that have been created but not yet destroyed in the system. In this section we illustrate the rules for transforming UML type structures to Z specifications that characterize configurations.

2.1 A formalization of types

A type, like a UML class, has a name, and consists of a set of attributes and operation specifications. Graphically, it is depicted as a rectangular box with three compartments: the top compartment contains the type name, the middle compartment contains the set of attributes (with optional types and initial values), and the third compartment contains the list of operations (with optional argument lists and return types).

The set of all instances of a type in a configuration is called the *state* of the type. This set is to be distinguished from the set of all *possible* instances that satisfy the type properties. Such a set is called the *type space* of the type. A type state must be a subset of the type space. When interpreted in isolation, a type denotes its type space. When interpreted in the context of a Type Diagram, a type denotes a type state.

The state of an instance consists of two components: a data state and a set of operation states. The data state of an instance consists of attribute and association values. The attribute values are the values associated with type attributes, and association values are the instances that are linked to the instance under consideration. Associations will be discussed in the next section. In a state of an instance, each operation is associated with an operation state of the form $< before_data_state, after_data_state(inputs) >$. The *before_data_state* is the (current) data state of the instance and the *after_data_state(inputs)* is the data state that is attained when the operation is performed to completion with inputs *inputs*.

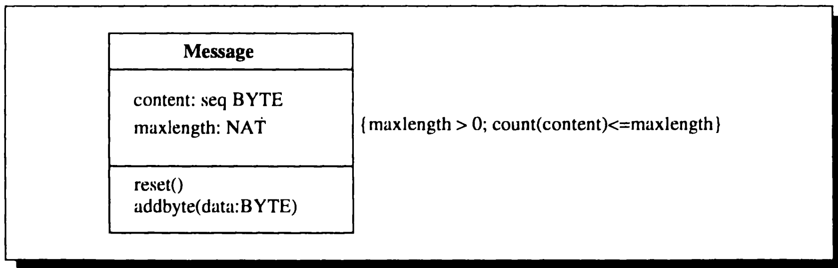

Figure 1 Message Type

In our formalization, a UML type is associated with a Z basic type consisting of elements representing unique instances of the UML type (they can be thought of as object identifiers). The attributes of a type are defined in a Z schema, referred to as an *attribute schema*. UML type invariants are expressed as predicates in the predicate part of attribute schemas. The attribute schema for the *Message* type shown in Fig. 1 is given in EXAMPLE 1. The type oper-

ation *reset* clears the message contents, and *addbyte* appends a byte of data to the message.

Example 1 Attribute schema for *Message* type

$[BYTE]$

Message_Attributes

$content : \text{seq}\ BYTE$
$maxlength : \mathbb{N}$

$maxlength > 0$
$\#content \leq maxlength$

Type operations are specified by Z schemas, called *operation schemas*, that relate before-data-states to after-data-states. Our operation schemas differ notationally from the traditional Z operation schemas in that we use a variable to represent the before-state and another to represent the after-state. The operation schemas for the *reset* and *addbyte* operations are given in EXAMPLE 2. In the schemas, m represents a before-state, and m' an after-state.

Example 2 Operation specifications for *Message* type

Reset

$m, m' : Message_Attributes$

$m'.content = \varnothing$
$m'.maxlength = m.maxlength$

AddByte

$m, m' : Message_Attributes$
$data? : BYTE$

$\#(m.content) < m.maxlength$
$m'.content = m.content \frown \langle data? \rangle$
$m'.maxlength = m.maxlength$

Semantically, an instance in a configuration can be viewed as a mapping of its object identifier to its data and operation states. This is captured formally by a Z schema that declares a variable representing the instances, and functions that map instances to their states. Such a schema is called a *type schema*. The type schema for the *Message* type is given in EXAMPLE 3 ($[MESSAGE]$ is supposed defined). The first three predicates of the *Message* type schema state that only configuration instances (elements of *instances*) are associated with data and operation states. The fourth predicate in the schema states that the before-state of an operation (m) must be the (current) data state of the instance (as determined by the function *attributes*).

Example 3 The type schema for *Message*

$$
\begin{array}{l}
\textit{Message} \\
\hline
\textit{instances} : \mathbb{P}\,\textit{MESSAGE} \qquad \text{[set of existing instances]} \\
\textit{attributes} : \textit{MESSAGE} \pfun \textit{Message_Attributes} \\
\textit{reset} : \textit{MESSAGE} \pfun \mathbb{P}\,\textit{Reset} \\
\textit{addbyte} : \textit{MESSAGE} \pfun \mathbb{P}\,\textit{AddByte} \\
\hline
\operatorname{dom} \textit{attributes} = \textit{instances} \\
\operatorname{dom} \textit{reset} = \textit{instances} \\
\operatorname{dom} \textit{addbyte} = \textit{instances} \\
\forall\, p : \textit{instances};\ \textit{att1} : \textit{Reset};\ \textit{att2} : \textit{AddByte} \\
\quad \mid \textit{att1} \in \textit{reset}(p) \land \\
\quad \textit{att2} \in \textit{addbyte}(p) \bullet \textit{att1}.m = \textit{attributes}(p) \\
\qquad \land\ \textit{att2}.m = \textit{attributes}(p)
\end{array}
$$

2.2 Formalization of UML associations

Semantically, an association is a set of *links*, where a link is a pair of instances of the form $(a \mapsto b)$, indicating that a and b are linked.

Multiplicity of an association constrains how many instances of a type can be associated with one instance of another (or the same) type. A range multiplicity is of the form $m..n$, where m is the lower bound and n is the upper bound of the range. The range $m..m$ can be simply written m. The multiplicity symbol '*' indicates many i.e. an unlimited number of objects. By itself, the symbol '*' is equivalent to '0..*' i.e. zero or more.

When associations are present, the data state of a type instance includes the instances that are related to the type. At the analysis level associations are bidirectional, that is, each linked instance knows about the instances it is linked to. This implies that each type instance includes information about its linked instances in its data state. Decisions related to restricting visibility of linked instances are best made during the design phase.

Consider the Type Diagram for a library system consisting of a *Copy* type and a *Borrower* type, with a many-to-one *Borrowed_by* association, shown in Fig. 2. The *Borrowed_by* association has an attribute *due_date* and a multiplicity that restricts a borrower to a maximum of 5 copies.

The formalization of the Type Diagram shown in Fig. 2 is given in EXAMPLE 4.

The association schema *Borrowed_by* defines the association as a set of pairs (*Rel*). The states of instances of the types *Borrower* and *Copy* are defined in

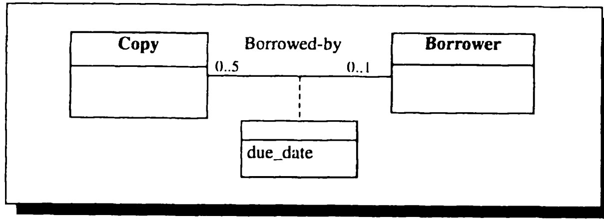

Figure 2 Example of an association with attributes

Example 4 The type schema for the *Borrowed_by* association

$[DATE, BORROWER, COPY]$

Borrowed_by_Attributes

$due_date : DATE$

Borrowed_by

$Rel : COPY \nrightarrow BORROWER$
$Rel_Attributes : (COPY \times BORROWER) \nrightarrow Borrowed_by_Attributes$

$\mathrm{dom}\ Rel_Attributes = Rel$
$\forall b : \mathrm{ran}\ Rel \bullet \#(Rel \rhd \{b\}) \leq 5 \land \#(Rel \rhd \{b\}) \geq 0$ [multiplicity constraint]

Borrower

$instances : \mathbb{P}\ BORROWER$
$borrowed_by : Borrowed_by$

$\mathrm{ran}\ borrowed_by.Rel \subseteq instances$

Copy

$instances : \mathbb{P}\ COPY$
$borrowed_by : Borrowed_by$

$\mathrm{dom}\ borrowed_by.Rel \subseteq instances$

AssocStruct

$b : Borrower$
$c : Copy$

$b.borrowed_by = c.borrowed_by$

the type schemas with the respective type names. The schema *AssocStruct* is a formalization of the Type Diagram shown in Fig. 2.

2.3 Formalization of Aggregation

An aggregate structure is a special type of association indicating a conceptual whole-part relationship. UML provides a weak and a strong form of aggregation. The strong form of aggregation is called a *composition*. An aggregate structure is depicted as an association of types in which a diamond is placed on the end of the association connected to the type that is the whole. If the diamond is filled then it is a composition, implying that each part can belong to only one whole, and that the lifetime of the parts are "coincident" with the lifetime of the whole (page 47, section 4.23 in (Booch *et al.*, 1997)). Such lifetime binding is not implied by the weak form of aggregation†. An unfilled diamond represents a weak aggregation, in which sharing of parts is allowed. For a composition, the multiplicity at the whole end must be no greater than 1 (restricting parts to belong to at most one whole). For the weak aggregation a multiplicity greater than one is allowed.

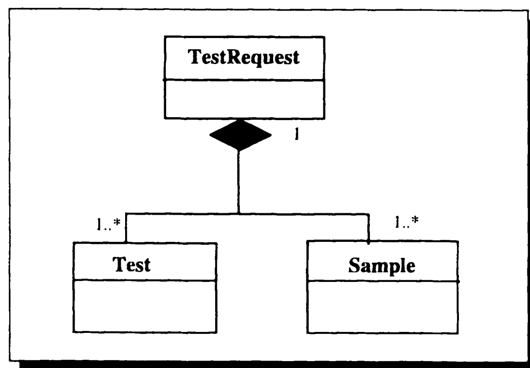

Figure 3 Example of a composition.

Below we formalize the composition shown in Fig. 3:

$[TEST, SAMPLE, TESTREQUEST]$

Test
$instances : \mathbb{P}\ TEST$

Sample
$instances : \mathbb{P}\ SAMPLE$

---

†To be more precise, the UML manual does not indicate that there is a lifetime binding of parts to whole in a weak aggregation.

TestRequest

$$
\begin{array}{l}
instances : \mathbb{P}\, TESTREQUEST \\
comp1 : TESTREQUEST \leftrightarrow TEST \\
comp2 : TESTREQUEST \leftrightarrow SAMPLE \\
\hline
\mathrm{dom}\, comp1 = instances \\
\mathrm{dom}\, comp2 = instances \\
\forall t : \mathrm{ran}\, comp1 \bullet \#(comp1 \rhd \{t\}) = 1 \\
\forall t : \mathrm{ran}\, comp2 \bullet \#(comp1 \rhd \{t\}) = 1
\end{array}
$$

AggStruct

$$
\begin{array}{l}
tests : Test \\
samples : Sample \\
testrequests : TestRequest \\
\hline
\mathrm{ran}(testrequests.component1) \\
\quad = tests.instances \\
\mathrm{ran}(testrequests.component2) \\
\quad = sample.instances
\end{array}
$$

In the schema *TestRequest*, the components of the aggregate are specified as mappings from the instances of *TestRequest* to instances of the parts *Test* and *Sample*. The first and second predicates restrict part instances to instances in the configuration. The third and fourth predicates state that no two distinct instances of the type *TestRequest* can share parts. This restriction is not needed for weak aggregation in which the multiplicty at the whole-end is greater than one. The schema *AggStruct* formalizes the Type Diagram in Fig. 3. The instances of *Test* and *Sample* must be related to one *TestRequest* instance in a configuration (as specified by the multiplicty of 1 at the diamond end). The formalization of the weaker form of aggregation can be obtained by weakening the restrictions of the stronger form.

2.4 Formalization of Generalization/Specialization Hierarchies

A generalization-specialization hierarchy captures a supertype-subtype relationship between types. It is represented as a link from the subtype to the supertype, with a large hollow triangle at the supertype end.

The attribute structure of a generalization-specialization hierarchy is represented in Z by including the schemas defining the shared attributes in subtype attribute schemas. Our formalization covers the four combinations of generalization-specialization according to whether or not the supertype is abstract (i.e., all instances are instances of some subtype in the model) or the subtypes are disjoint (i.e., the subtypes do not share instances).

3 A FORMALIZATION EXAMPLE

In this section we illustrate the application of the UML-to-Z rules outlined in the previous section on a small, but non-trivial Type Diagram.

A Type Diagram for a Glyph structure is shown in Fig. 4. The complexity of recursive structures often causes modelers to underspecify their desired

properties. Formalizing such structures forces modelers to consider and express required constraints that may have been glossed over in a less formal approach. Once a formal model is obtained, it can be used to demonstrate that the desired properties are present.

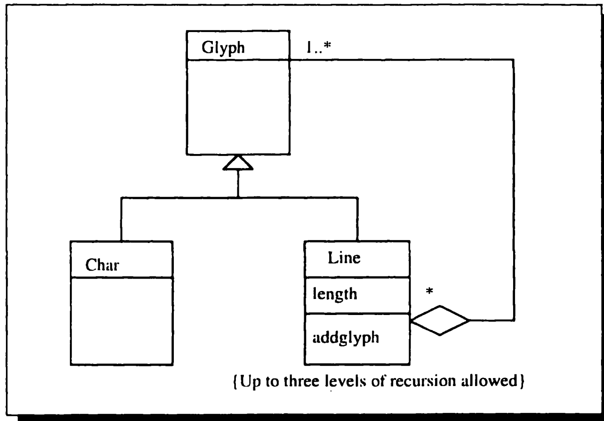

Figure 4 A recursive UML Type Diagram

A glyph is defined to be an abstract type for objects that can appear in a document structure. Its subtypes are the primitive graphical character elements (elements of type *Char*) and structural line elements (elements of type *Line*). This technique of composing increasingly complex elements out of simple ones in a hierarchical fashion is called *recursive composition.*

The type schemas for *Glyph* and *Char* are given below:

$[GLYPH]$

Glyph
$instances : \mathbb{P}\, GLYPH$

Char
$instances : \mathbb{P}\, GLYPH$

The recursive composition is constrained as follows (the 'part-of' aggregation relationship is transitive):

- The aggregation is anti-symmetric (i.e., if line $l1$ is a part of line $l2$ then $l2$ cannot be a part of $l1$).
- The aggregation is irreflexive (i.e., a line cannot be a part of itself).
- No more than three levels of nesting is allowed (a line $l1$ can contain a line $l2$ that contains a line $l3$; $l3$ must consist only of characters). This is an application-specific constraint (see annotation on diagram).

The type schema for *line* is given below:

GlyphComp

$$aggrel : GLYPH \leftrightarrow GLYPH$$

$$\forall l1, l2 : GLYPH \mid (l1, l2) \in aggrel^{+} \bullet (l2, l1) \notin aggrel^{+}$$

$$[aggrel^{+} \text{ is the transitive closure of } aggrel]$$

$$\forall l : GLYPH \bullet (l, l) \notin aggrel^{+}$$

$$\forall l1, l2, l3, l4 : GLYPH \mid (l1, l2) \in aggrel \land (l2, l3) \in aggrel \bullet (l3, l4) \notin aggrel$$

Line_Attributes

$$length : \mathbb{N}$$

$$length > 0$$

AddGlyph

$$l, l' : Line_Attributes$$
$$c, c' : GlyphComp$$
$$g? : GLYPH$$

$$l'.length = l.length + 1$$
$$c' = c \cup \{g?\}$$

Line

$$instances : \mathbb{P}\, GLYPH$$
$$attributes : GLYPH \rightarrowtail\!\!\!\!\!\!\!\!\;\;+\; Line_Attributes$$
$$components : GlyphComp$$
$$addglyph : GLYPH \rightarrowtail\!\!\!\!\!\!\!\!\;\;+\; \mathbb{P}\, AddGlyph$$

$$\operatorname{dom} attributes = instances \land \operatorname{dom}(components.aggrel) = instances$$
$$\forall g : instances \bullet addglyph(g).l = attributes(g) \land addglyph(g).c = components$$

Using the schemas defined above, the following Z formalization of the Type Diagram in Fig. 4 is obtained:

GlyphStruct

$$glyphs : \mathbb{P}\, Glyph$$
$$lines : \mathbb{P}\, Line$$
$$chars : \mathbb{P}\, Char$$

$$\langle lines.instances, chars.instances \rangle \; partition \; glyphs.instances$$
$$\forall l : lines \bullet \operatorname{ran}((l.components).aggrel) \subseteq glyphs.instances$$

The first predicate in *GlyphStruct* states that the supertype is abstract and the subtypes are disjoint. The second predicate states that lines are composed of existing glyphs.

The benefit of having a formal model of the recursive structure is that one can prove properties that cannot be demonstrated simply be examining the diagram (e.g., the property that up to 3-levels of nesting is allowed). Building the formal model also forces one to consider, in detail, the constraints that

are needed. Formalizing and analyzing UML structures can lead to a better understanding of the modeled structure.

4 CONCLUSION AND FUTURE WORK

The formalization of UML models can lead to a deeper understanding of modeled structure, and allows one to rigorously reason about modeled properties. In this paper we have illustrated how formalization can help clarify the meaning of non-trivial structures such as recursive definitions[‡]. Recursive structures have always been complex to model and to analyze. The lack of firm semantic bases for OO models compounds the complexity problem by increasing the chances of introducing ambiguous, incomplete, and imprecise statements of desired behavior and structure.

We have provided a semantic model that supports the formal interpretation of complex UML type structures, and the rigorous analysis of modeled properties. The Z specifications derived from the semantic model are tedious to produce by hand. Mechanical support for the generation of Z specifications from UML type structures is essential and possible. We are currently extending a tool we built for generating Z specifications from Fusion Object Models (see (France *et al.*, 1997)) to support the UML-to-Z transformation.

REFERENCES

Booch, Grady, Rumbaugh, James, & Jacobson, Ivar. 1997 (Jan.). *Unified Modeling Language.* Version 1.0. Rational Software Corporation, Santa Clara, CA-95051, USA.

Bowen, Jonathan P., & Hall, J. Anthony (eds). 1994. *Z User Workshop, Cambridge 1994.* Workshops in Computing. Springer-Verlag, New York.

Bruel, Jean-Michel, France, Robert B., & Benzekri, Abdelmalek. 1996 (21–25 Oct.). A Z-based Approach to Specifying and Analyzing Complex Systems. *In: Proceedings of the Second IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'96), Montreal, Canada.*

France, Robert B., Bruel, Jean-Michel, & Larrondo-Petrie, Maria M. 1997. An Integrated Object-Oriented and Formal Modeling Environment. *To appear in the Journal of Object-Oriented Programming (JOOP).*

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

[‡]The authors wish to thank the members of the Methods Integration Research Group (MIRG) for their participation on this project. For more information on MIRG, and integrated formal and OO modeling techniques see the WWW site at: *http://www.cse.fau.edu/research/MIRG/.* This work was partially funded by NSF grant CCR-9410396.