A Formal Semantics for Object Model Diagrams

Robert H. Bourdeau and Betty H.C. Cheng, *Member, IEEE Computer Society*

***Abstract*—Informal software development techniques, such as the *Object Modeling Technique* (OMT), provide the user with easy to understand graphical notations for expressing a wide variety of concepts central to the presentation of software requirements. OMT combines three complementary diagramming notations for documenting requirements: *object models, dynamic models*, and *functional models.* OMT is a useful organizational tool in the requirements analysis and system design processes. Currently, the lack of formality in OMT prevents the evaluation of completeness, consistency, and content in requirements and design specifications.**

A formal method is a mathematical approach to software development that begins with the construction of a formal specification describing the system under development. However, constructing a formal specification directly from a prose description of requirements can be challenging. This paper presents a formal semantics for the OMT object model notations, where an object model provides the basis for the architecture of an object-oriented system. A method for deriving modular algebraic specifications directly from object model diagrams is described. The formalization of object models contributes to a mathematical basis for deriving system designs.

***Index Terms*—Algebraic specification, formal methods, object modeling, requirements specification, semantics.**

I. Introduction

SOFTWARE requirements define the objectives of a software development effort. Requirements provide the basis by which the quality of the end-product is measured and should guide the system design. Recent studies have shown that the introduction of errors to software is most likely to occur during the early phases of requirements analysis and design; these errors can have a lasting impact on the reliability, cost, and safety of a system [1], [2]. Furthermore, requirements errors are between 10 and 100 times more costly to fix during later phases of the software lifecycle than during the requirements phase [3], [4], [5]. These data illustrate the need to develop analysis methods that enable a rigorous assessment of requirements.

One approach to this problem is to document software requirements and design using a formal language; such a document is called a *formal specification* [6]. This approach is one of the basic elements of the software engineering disciplines referred to as *formal methods.* A formal method is characterized by a formal specification language and a set of rules governing the manipulation of expressions in that language [6]. While the advantages to using formal methods are significant, including the use of notations that are precise, verifiable, and facilitate automated processing [6], [7], attempting to construct a formal specification directly from a requirements document can be challenging. Formal descriptions potentially involve considerable syntactic detail and require careful planning and organization on the part of the specifier in order to obtain modular specifications. Minor changes in the natural language description, or the interpretation thereof, can require significant and tedious reorganization of the formal description.

Other approaches to requirements analysis and design include the numerous object-oriented techniques [8], [9], [10], [11]. These "informal" methods enable the rapid construction of a system model using intuitive graphics and user-friendly language. While such techniques are recognized as useful tools, the graphical notations used with these methods are often ambiguous, resulting in diagrams that are easily misinterpreted. In order to ensure the reliability of systems developed with these informal approaches, complex validation procedures must be incorporated at every stage of development, and extensive testing methods must be used [12].

One object-oriented development approach, the *Object Modeling Technique* (OMT) [10], uses three types of models to express important domain-related concepts: object models, functional models, and dynamic models. Each model contributes to the understanding of a system, but the object model is of central importance. Those elements of a system that define its static structure are given by an *object model* using a notation similar to that used for entity-relationship diagrams [13]. An object model determines the types of objects that can exist in the system and identifies allowable relationships among objects. As a result, the object model constrains the set of possible states that the system may enter. A *dynamic model* describes valid changes to system states and indicates the conditions under which a state change may occur. The notation used for dynamic models is a variation of Harel's Statechart notation [14] which, in turn, is an extension to the traditional notation for finite automata [15]. A *functional model* is a data flow diagram that describes the computations to be performed by the system. The main attractive feature of the OMT approach is that by using these three notations in a complementary manner, the system developer can express and refine system requirements into a design and implementation. However, the lack of precise definitions for the notations makes it difficult to combine this approach with rigorous, systematic software development methods.

Rumbaugh et al. [10] define the semantics of an object model, in part, by displaying a series of *instance diagrams* as examples of what a specific object model describes. In this paper, we make extensive use of instance diagrams and define the state space of an object model as the set of all such instance diagrams of that object model. We show that an object

Manuscript received February 1994; revised January 1995.

R. Bourdeau is currently with the Consortium of International Earth Science and Information Network, Saginaw, Michigan; e-mail: bourdeau@ciesin.org. This work was done while he was a PhD student at Michigan State University.

B. Cheng is with the Department of Computer Science, Michigan State University, East Lansing, MI 48824; e-mail: chengb@cps.msu.edu.

IEEECS Log Number S95041.

model diagram S corresponds to an algebraic specification $SPEC\ (S)$ [16], and that an instance diagram I of S corresponds to an algebra that satisfies $SPEC\ (S)$. The formalization process is presented as a method for deriving algebraic specifications from object model diagrams. The formal description of the architecture of a system and its allowed system states can be used to facilitate the design and implementation stages as well as enable the use of rigorous techniques for verification and validation tasks.

The remainder of this paper is organized as follows. Section II provides an introduction to the object model notation, including an informal discussion of the semantics of the notation. Section III describes the methodology used in the formalization process, which includes a bottom up construction of the semantics of object models. Section IV compares the results of this paper with related work. Finally, Section V summarizes the major results of this paper and describes ongoing and future investigations.

II. Object Model Syntax

This section presents the basic notation used to represent object models. An object-oriented distributed multimedia decision support system that is being developed for accessing and analyzing environmental science and global change information is introduced and will be used to illustrate the object model formalization throughout the paper.

A. Basic Syntactic Elements

The kernel of the object-oriented "paradigm" is a description of the structure of the "world" in which a system operates. This "world," more commonly referred to as the *problem domain*, is the foundation upon which functional and operational requirements are expressed. An example that provides a useful illustration of problem domain analysis and will be used to demonstrate many concepts throughout this paper is a distributed information system. The system, ENFORMS (Environmental Information System), is an existing system being developed for NASA, USDA, and EPA that is used to establish a dynamic network of distributed earth-science data archives with integrated on-line access services [17], [18]. The problem domain in which ENFORMS operates consists of a set of data centers that may be physically distributed throughout a geographic region. ENFORMS allows these data centers to make their data available through a regional data sharing network (established by ENFORMS) and provides a variety of tools (a mapping utility, for example) for locating and manipulating that data. The most general requirements that guide the problem analysis are as follows.

1) Data centers must be able to independently manage both content and access mechanisms for their data stores.
2) Data centers may disable access to their data stores at any time (i.e. go off-line).
3) Data centers may make their data stores available at any time (i.e. go on-line).
4) At any instant, all on-line data centers must be available to any ENFORMS user.

These requirements impose several constraints on the design of ENFORMS.

Fig. 1 gives a high-level object model for the ENFORMS problem domain. The OMT notation uses boxes to depict the *types* of objects that can exist in the system, more commonly called *classes*. Four classes are shown in Fig. 1: *User*, *Dataset*, *Network Manager*, and *Data Center*. Each of these classes, excluding the Network Manager class, correspond to key nouns in the general ENFORMS requirements. The Network Manager class is introduced to allow the notion *on-line* to be expressed. As a presentation convention, when referring to the formal concept represented by a class, the name of the class is capitalized (Dataset, for example).

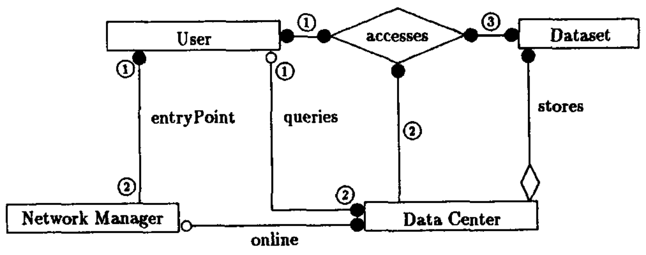


Fig. 1. A high-level object model, O_1, for a distributed information system.

Relationships between objects, known as *associations*, are specified by lines interconnecting the classes of a model. The simplest association is a general binary association and is depicted by a labeled line. Fig. 1 shows three such associations, the lines labeled *entryPoint*, *queries*, and *on-line*. Associations cannot always be written so that the name of the association allows a left-to-right reading. The numbers enclosed in small circles labeling each end of an association provide an orientation for reading. The orientation of an association helps in understanding how the name of the association is read with respect to the classes it relates. A more important purpose of the numbering notation is to allow the formal syntax of the association to be derived unambiguously. Formally, an association is a predicate whose arguments are objects from the classes it relates. In the formalization, the order of the arguments is given by the association's *signature* (syntax structure), and the numbering notation shown in the diagram determines this order.

Two other associations are also depicted, *stores* and *accesses*. The association labeled *stores* is an aggregation association, indicating that a *Data Center* is an aggregate (a composition) of *Dataset* objects. *Accesses* is an association of arity three, and thus uses the diamond as a joining point for the classes it associates; hence, this association specifies that a User accesses a Dataset via a Data Center.

In addition to specifying the names of the relationships that exist between objects, constraints on those relationships are also given by an object model. The simplest constraint is called a *multiplicity constraint*, which pictorially corresponds to the endpoints of a line depicting an association. A multiplicity constraint describes a restriction on the number of objects from one class that may be associated with any one object from another class. For example, a solid circle endpoint, shown in Fig. 1, indicates a multiplicity of *many*.

Therefore, the *entryPoint* association specifies that, at any time, a Network Manager serves as the entry point (to an ENFORMS network) for "zero or more" (many) users. The end of the *entryPoint* association opposite the solid circle also has an endpoint, a straight line. The straight line indicates a multiplicity of "exactly one," and specifies that a user may enter an ENFORMS network via a single Network Manager. The hollow circle endpoint indicates a multiplicity of "zero or one". In Fig. 1, this endpoint is used to specify that a Data Center may be on-line with at most one Network Manager.

The simple object model given in Fig. 1 conveys a significant amount of information about possible *instances* of an ENFORMS system. One such instance (I_1) is depicted graphically in Fig. 2. In this figure, large circles depict objects and dotted arrows connect the objects to their respective classes. For example, objects d_1, d_2, and d_3 represent Data Centers. Data Centers d_1 and d_3 are being *queried* by User u, but d_2 is being *queried* by no User. This type of pictorial representation of an object model instance is called an *instance diagram*, which gives an abstract representation of a system's state.

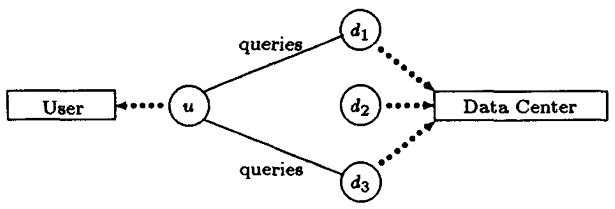


Fig. 2. A simple instance diagram, I_1.

Fig. 3 depicts a more complex instance diagram for the object model given in Fig. 1. Here, a Network Manager, n, serves as an entry point to an ENFORMS network for two Users, u_1 and u_2. One Data Center, d, is on-line with the Network Manager n, and d stores two Datasets, i_1 and i_2. User u_1 is accessing Dataset i_1 through Data Center d, and User u_2 is accessing no Datasets. Dataset i_2 is not being accessed at all. This scenario can be shown to be a valid state for an ENFORMS system, as specified by the object model given in Fig. 1.

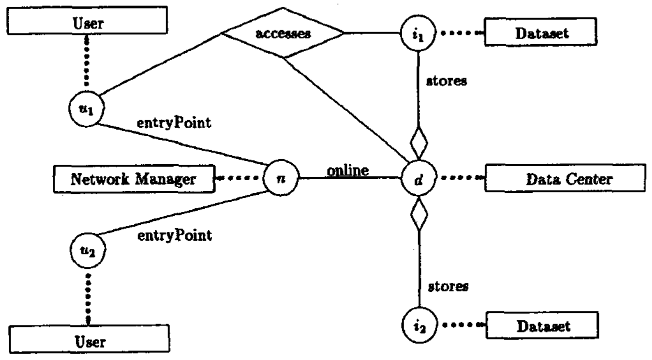


Fig. 3. A more complicated instance diagram.

It is also possible to graphically depict states that are not consistent with an object model. The instance diagram, I_2, given in Fig. 4 shows such a scenario. Here, two different Network Managers, n_1 and n_2, are acting as entry points to an ENFORMS network for a single user, u. This situation is specifically prohibited by the ENFORMS object model, as specified by the one-to-one relationship between the Network Manager and User classes.

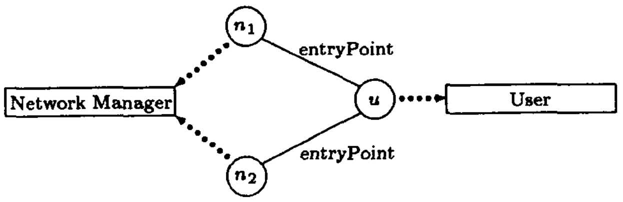


Fig. 4. An inconsistent instance diagram, I_2

B. Object States and Attributes

For any given object, there are a variety of characteristics that might be important to the ongoing activities of a system. For example, in order to properly manipulate georeferenced data it is necessary to know the scale of the data and the units of measurement. Such information does not require revealing the internal structure of an object, but can be provided as *attribute* or *state* information.

Information about the state of an object is modeled as a *valuation*. An object may have many different valuations, each one providing a different kind of observation about the object in question. The simplest valuation is the *state valuation*, which identifies the state of an object. For example, a Data Center might be in one of three states: *off-line*, *busy*, and *available*. This set of possible states is depicted in Fig. 5. The double-headed arrows leading away from the class Data Center specify possible states for objects of that class. We refer to these states as *object-states* as they represent the state of the object to which they are linked.

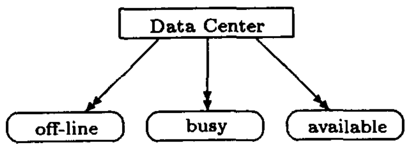


Fig. 5. Object model notation for defining object-states.

The object-state notation is an extension to the object model notation defined within OMT. Furthermore, OMT object models use ovals to depict *objects* instead of states. Since ovals are used in the OMT dynamic model to represent states, the same graphical notation in the object model provides a visual linkage between the two models. It should also be noted that our semantics for the object-state notation can be applied consistently to object models that do not define states; the notation is thus a true extension. The object-state is the simplest of the possible observations that can be defined over a class of objects.

The last syntactic element to be discussed in this section is the notation for specifying object attributes. Attributes are observable characteristics of an object. Object-states are the simplest kind of attribute, as they provide a simple summary of the condition of an object. A more general notion of attributes is one that allows different kinds of observations, each one given a name.

In the traditional OMT notation, attributes are listed within the box that encloses the name of the class being described. For example, the class Document depicted in Fig. 6 uses the traditional OMT notation to name five attributes: *author*, *dte*, *title*, *content*, and *abstract*. One potential problem with this notation is that it may suggest a design decision about the class Document, in that it appears that the five attributes are data internal to the class. A second problem with this notation is that, for more complicated classes, the amount of textual information placed inside the box becomes quite cumbersome and does not explicitly represent the dependencies between the named class, and the classes named as attribute types (e.g. Text, Date, and Name).

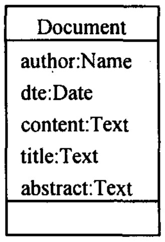


Fig. 6. Traditional OMT notation for attributes.

Our alternative to the traditional OMT notation for expressing attribute information is a graph-based syntax. The information depicted in Fig. 6 can be represented using our syntax as depicted in Fig. 7. An attribute specification is a labeled, double-headed arrow connecting two classes. The arrow is labeled with the name of the attribute, the class at the head of the arrow is the attribute's type, and the class at the tail is the class that the attribute is describing. We argue that the same information is being depicted, but in a less implementation-biased manner. An attribute is simply the name of an observation that can be made about any object belonging to the indicated class.

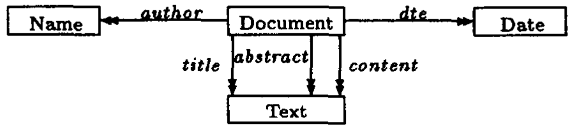


Fig. 7. Our graph-based notation for attributes.

C. Subtyping

A basic principle of object-oriented analysis and design is to identify observable similarities among the objects that exist in a system. Objects are classified according to similarities, and the specifier describes a system in terms of these classes of objects rather than specific objects. Two classes may also share similarities, and these similarities form the basis of a class hierarchy. The primitive relationship that characterizes a class hierarchy is the *subtype* relationship.

In the literature, it is generally agreed that a class X is a *subtype* of a class Y when the objects of X are indistinguishable from Y-objects [19], [20]. Object-oriented development methods typically incorporate some concept of subtyping in their methodology, but it is not always well-defined. We provide a formal characterization of subtyping in the semantics of object models.

Graphically, a subtype relationship between two classes is depicted as a decorated line connecting the two classes. The line is decorated by a triangle that points in the direction of the supertype. On the opposite end of the line, the subtype is connected. In Fig. 8, the classes RequestMsg and ReplyMsg are asserted to be subtypes of the class Message. The class Message is shown as having a Transaction Number as a component. Since RequestMsg and ReplyMsg are subtypes of Message, they must have all the properties of Message; hence, RequestMsg and ReplyMsg must have a Transaction Number component as well. Using the terminology of object-orientation, RequestMsg and ReplyMsg *inherit* the Transaction Number component of the supertype Message.

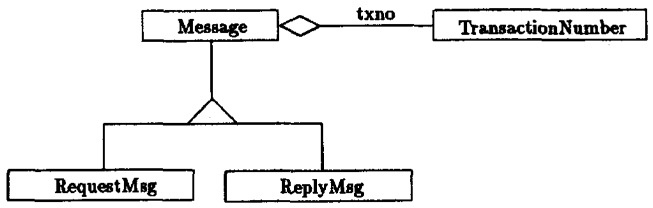


Fig. 8. An example of subtyping.

Fig. 8 shows the starting point for the specification of a message passing protocol that is the basis of interaction between the Network Manager, User, and Data Center in the ENFORMS system. Fig. 9 illustrates a more detailed class hierarchy for messages; specifically, the subtypes of RequestMsg are depicted. The immediate subtypes of RequestMsg partition the types according to the entities involved in the request. Each class name consists of a mnemonic name for the type of sender, an arrow to suggest the sending operation, and a mnemonic name for the type of receiver (CL refers to the user (<u>cl</u>ient), NS refers to the network manager (<u>n</u>ame <u>s</u>erver), and AS refers to the the data center (<u>a</u>rchive <u>s</u>erver)). For example, all message types that are subtypes of CL$\longrightarrow$AS correspond to message objects sent from a user (client) to a data center (archive server).

III. Object Model Semantics

In this section, an algebraic formalization of the semantics for the object model and instance diagram notations is given. The key concepts of this formalization process are highlighted in Fig. 10. The arrow labeled "OMT semantics" represents the relationship between object models and instance diagrams that has been described by Rumbaugh et al. [10]. Our approach proposes that instance diagrams *can* be used to provide a

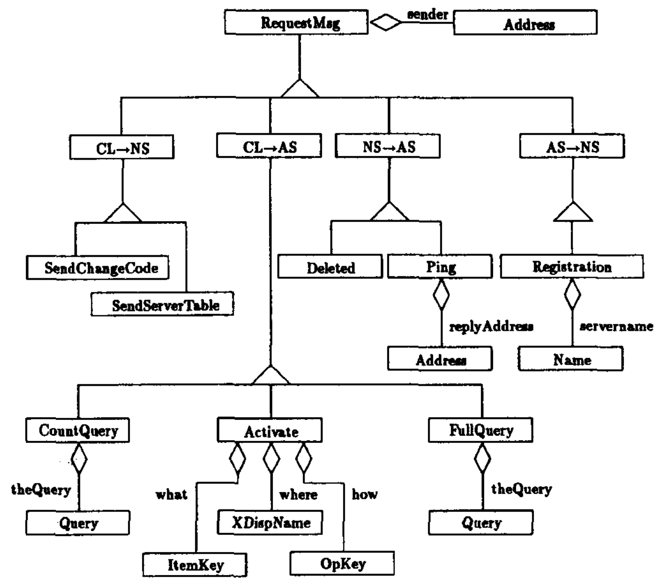


Fig. 9. The class hierarchy of RequestMsg types.

graphical definition of semantics for object models. This formalization relies on the existing work on algebraic specification that has been well developed in the literature [16], [21], [22]. In our approach, object models are formalized as algebraic specifications, and instance diagrams are formalized as algebras. This step corresponds to the two arrows labeled "formalized as" in Fig. 10.

$$\begin{tikzcd}
\text{object models} \arrow[r, "\text{OMT semantics}"] \arrow[d, "\text{formalized as}"'] & \text{Instance Diagrams} \arrow[d, "\text{formalized as}"] \\
\text{Algebraic Specifications} \arrow[r, "\text{algebraic semantics}"'] & \text{Algebras}
\arrow[from=1-1, to=2-2, phantom, "\textit{indirectly formalized as}"]
\end{tikzcd}$$

Fig. 10. Basic approach to formalization.

Algebraic specifications have algebras as their semantics. This relationship is represented in the diagram by the arrow labeled "algebraic semantics". We argue that the diamond diagram in Fig. 10 "commutes". That is, the semantics of an object model is a set of algebras. There are two ways to determine these algebras, as shown in the diamond diagram: compute the algebraic specification of the object model and look at a set of algebras that satisfy this specification, or, determine the set of instance diagrams consistent with the object model and compute their corresponding algebras. If the diamond diagram given in Fig. 10 commutes, then either method for determining the semantics of an object model will yield the same set of algebras. As a result, the OMT semantics, which were previously imprecisely defined, are now described rigorously in terms of an algebraic semantics.

The remainder of this section is organized as follows. Section III-A gives the semantics of classes, objects, and attributes. Sections III-B and III-C describe the semantics of binary and n-ary associations, respectively. In Section III-D, the meaning of the subtyping notation is formalized, and in Section III-E all of the formalizations are combined in order to describe the semantics of object models.

A. Classes, Objects, and Attributes

We use the *Larch Shared Language* (LSL) [22] to illustrate how these basic formalisms are incorporated into a structured, algebraic specification. Larch was chosen as the target specification language because it has a simple syntax and there are a number of tools that can be used to manipulate the specifications, including a syntax checker [23], the Larch (theorem) prover [24], graphical browser and development environment [25]. LSL uses specification modules, called *traits*, to describe abstract data types and theories. Traits are presented in the following form:

SPECNAME (*parameters*) : trait
  includes
    existing specification modules to be used
  introduces
    function signatures are listed here
  asserts
    axioms are listed here

SPECNAME is the name of the specification module. The sorts that are to be considered as the parameters of the module are given in the parameter list following the name of the trait. *Includes* indicates other traits upon which the given trait is built. The *introduces* section itemizes *function signatures*, each of which gives the number and types of input arguments and result type of a function. *Asserts* defines the constraints for the specification. When using LSL, one assumes that a basic axiomatization of Boolean algebra is a part of every trait. This axiomatization includes the sort *BOOL*, the Boolean constants *true* and *false*, the connectives '$\wedge$' and '$\vee$,' implication '$\Rightarrow$,' and negation '$\neg$'.

Let O be an object model, and let $C = \{C_1, \ldots, C_n\}$ be the set of class names given in O. Formally, each class name $C_i \in C$, where $1 \le i \le n$, is considered to be the name of a *sort.* Every class C_i is asserted to have a special object err_{C_i} called an *error object.*

$$err_{C_i} : \to C_i$$

For each class name C_i, we introduce a sort C_i–*STATES*, which characterizes the set of states that are possible for any C_i-object. For each object-state s of class C_i, s is specified with the signature

$$s : \to C_i - STATES$$

Object-states are nullary functions with no input arguments; therefore, they are considered as constants. For every class C_i, we require that the set of possible states defined by C_i–*STATES* include a state $undef_{C_i}$; this state allows the state evaluation function to be total, yet still indicates when a C_i-object is in an *undefined* state. The corresponding signature is

$$undef_{C_i} : \to C_i - STATES,$$

which specifies $undef_{C_i}$ as a nullary function, and hence a constant in C_i–*STATES*. Each object-state s of a class C_i must be a distinct element of C_i–*STATES*. Uniqueness is ensured by asserting that for every pair of object-states s_1 and s_2 of C_i (including $undef_{C_i}$),

$$s_1 \neq s_2.$$

Furthermore, the set of object-states $s_1, \ldots, s_k$ for a given class C_i, in addition to the undefined state, completely define the sort C_i–*STATES*. This condition is indicated by adding the clause

$$C_i - STATES \ \textbf{generated by} \ s_1, \ldots, s_k, \ undef_{C_i}.$$

In order to bind a C_i-object to one of its possible states, we introduce a *valuation function*, \$, with the signature

$$\$: C_i \rightarrow C_i\text{–}STATES,$$

for each class name $C_i \in C$. The state of the error object is always undefined, and an axiom is added to describe this condition:

$$\$\left(err_{C_i}\right) = undef_{C_i}.$$

The definition of each valuation function varies with the system state, so no additional constraints are required.

Consider the following object model consisting of a single class *C*.

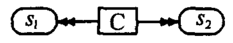
$$\boxed{C}$$

Since the set of class names *C* consists of a single class name, *C*, according to the description above, the specification has only two sorts: *C* and *C–STATES*. The only functions are the valuation function \$ on *C* and the two constants (constant functions) $undef_C$ and err_C. Hence, the simple specification given in Fig. 11 is derived.

```
EX1 : trait
    introduces
        undefC     :    → C-STATES
        errC       :    → C
        $          :  C → C-STATES
    asserts
        C-STATES  generated by undefC
        $(errC) = undefC
```

Fig. 11. A simple class specification.

Consider the following object model, O_3, containing object-states s_1 and s_2.



By applying the above formalizations to O_3, the specification given in Fig. 12 is obtained. Since there were three object-

```
EX2 : trait
    introduces
        errC       :    → C
        undefC     :    → C-STATES
        s1, s2     :    → C-STATES
        $          :  C → C-STATES
    asserts
        C-STATES  generated by s1, s2, undefC
        s1 ≠ s2
        s1 ≠ undefC
        s2 ≠ undefC
        $(errC) = undefC
```

Fig. 12. A simple class specification with states.

$$\begin{tikzcd}
\text{object model } \mathcal{O}_3 \arrow[r, "\text{has instance}"] \arrow[d, "\text{formalized as}"'] & \text{instance diagram } \mathcal{I}_6 \arrow[d, "\text{formalized as}"] \\
\text{spec } EX_2 \arrow[r, "\text{has algebra}"'] & \text{algebra } \mathcal{I}_6
\end{tikzcd}$$

Fig. 13. An example of the formalization process.

$C^{\mathcal{A}_1}$	$\stackrel{def}{=} \{e_1, e_2, ERRC\}$
$C\text{-}STATES^{\mathcal{A}_1}$	$\stackrel{def}{=} \{0, 1\}$
$\$^{\mathcal{A}_1}$	$\stackrel{def}{=} \{(e_1, 0), (e_2, 1), (ERRC, 0)\}$
$undef_C^{\mathcal{A}_1}$	$\stackrel{def}{=} 0$
$err_C^{\mathcal{A}_1}$	$\stackrel{def}{=} ERRC$
$s_1^{\mathcal{A}_1}$	$\stackrel{def}{=} 1$
$s_2^{\mathcal{A}_1}$	$\stackrel{def}{=} 1$

Fig. 14. A Σ_{EX_2}-algebra, named $\mathcal{A}_1$.

states in the object model, $undef_C$, s_1, and s_2, three inequality axioms are needed to establish uniqueness among object-states. In general, for *k* object-states, $\binom{k}{2}$ (*k choose 2*) axioms are needed to establish the uniqueness of each state.

Next, we discuss the relationships between algebraic specifications and algebras. Fig. 13 depicts the high-level process used to determine consistency of instance diagram I_6 and object model. Let *SPEC* be an algebraic specification (in LSL, a trait gives an algebraic specification). The *signature* Σ of *SPEC* is a 2-tuple 〈*SORTS,FUNS*〉, where *SORTS* is the set of sort names of *SPEC*, and *FUNS* is the set of function signatures of *SPEC*.[1] (The set *SORTS* comprises the sort names referenced in the '`introduces`' section of a trait, and *FUNS* comprises the function signatures given in the same section.) Given a specification *SPEC* with signature Σ = 〈*SORTS,FUNS*〉, a *Σ-algebra* defines a set of elements for each sort name in *SORTS* (called a *carrier set* [16]) and defines a function on these sets for each function symbol in *FUNS*.[2] If we let σ_{ex_2} represent the signature of the specification EX_2, then Fig. 14 defines a Σ_{ex_2}-algebra, which we refer to as algebra $\mathcal{A}_1$.

In Fig. 14, each symbol in the algebraic specification EX_2 is explicitly defined, as required by the definition of an algebra. Each function and sort symbol from the specification EX_2 is superscripted by $\mathcal{A}_1$, indicating that the given definition of that symbol is relative to algebra $\mathcal{A}_1$. The carrier sets for sorts *C* and *C–STATES*, relative to algebra $\mathcal{A}_1$, are denoted by $C^{\mathcal{A}_1}$ and $C-STATES^{\mathcal{A}_1}$, respectively. The signatures of the nullary functions s_1, s_2, and $undef_C$, relative to algebra $\mathcal{A}_1$ become

$$undef_C^{\mathcal{A}_1}, s_1^{\mathcal{A}_1}, s_2^{\mathcal{A}_1} : \rightarrow C - STATES^{\mathcal{A}_1},$$

and the definitions for these three functions are consistent with

1. This definition of FUNS allows overloading of the function names, but it is not the standard mathematical construction used in the literature [16].

2. This notation is standard in the literature on algebraic specification. A Σ-algebra is an algebra based only on a signature, Σ, and not on any axioms.

this signature. For example, $\mathcal{A}_1$ defines the constant function $s_1^{\mathcal{A}_1}$ to be the symbol '1'. Since $1 \in C-STATES^{\mathcal{A}_1}$, this definition for $s_1^{\mathcal{A}_1}$ is valid. The valuation function \$ is defined by algebra $\mathcal{A}_1$ as a set of n-tuples; if a 2-tuple (e, s) is in the set $\$^{\mathcal{A}_1}$, then $\$^{\mathcal{A}_1}(e) = s$.

Given a specification *SPEC* with signature Σ, we are interested in those Σ-algebras that define functions consistent with the axioms of *SPEC*; such an algebra is said to *satisfy SPEC*. As an example of the type of inconsistency that can be detected, consider algebra $\mathcal{A}_1$ that satisfies the specification EX_2. Relative to the algebra $\mathcal{A}_1$ given above, the axioms of EX_2 are

$$s_1^{\mathcal{A}_1} \neq s_2^{\mathcal{A}_1},$$
$$s_1^{\mathcal{A}_1} \neq undef_C^{\mathcal{A}_1},$$
$$s_2^{\mathcal{A}_1} \neq undef_C^{\mathcal{A}_1},$$
$$\$^{\mathcal{A}_1}\left(err_C^{\mathcal{A}_1}\right) = undef_C^{\mathcal{A}_1}.$$

By substituting the symbols for their definitions, as given in Fig. 15, we have the set of assertions

$$1 \neq 1,\ 1 \neq 0,\ 1 \neq 0,\ 0 = 0,$$

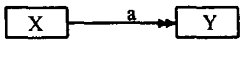
respectively. Clearly the first assertion is false. Therefore, while algebra $\mathcal{A}_1$ is a Σ_{ex_2}-algebra, it *does not* satisfy the specification EX_2. On the other hand, Fig. 15 gives a Σ_{ex_2}-algebra, $\mathcal{A}_2$, that *does* satisfy specification EX_2.

In general, any instance diagram can be denoted by an algebra. Consider, for example, the instance diagram I_6 given in Fig. 16a, which depicts two objects of class C, e_1 and e_2. Object e_2 is shown to be in state s_2, while the state of e_1 is undefined. Fig. 16b gives an algebra that denotes instance diagram I_6 and that satisfies specification EX_2. Of particular importance in this algebra (which we also name I_6 for brevity) are the definitions of the sort C and the valuation function \$. The definition of sort C is the carrier set C^{I_6}, which contains two elements corresponding to names of the objects given in the instance diagram. The valuation function $\$^{I_6}$ asserts that $\$^{I_6}(e_2) = s_2$, thus defining the state of object e_2 to be s_2, while the state of the object e_1 evaluates to the object-state $undef_C$. This default mapping allows \$ to be a total function even when no object-states are explicitly specified by the analyst.

As indicated in Fig. 13, the object model O_3 was formalized as the algebraic specification EX_2, and instance diagram I_6 was formalized as the algebra I_6. Hence, the claim of consistency between object model O_3 and instance diagram I_6 is reduced to a claim of consistency between the specification EX_2 and algebra I_6, and the latter claim can be proved rigorously.

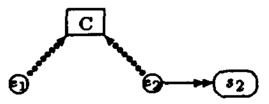
A trait that specifies a class C, using the rules described in this section, will have the trait name *CLASS-C*.

Recall the format used to graphically specify attributes for a class.



$$\begin{array}{ll}
C^{\mathcal{A}_2} & \overset{def}{=} \{e_1, e_2 ERRC\} \\
C\text{-}STATES^{\mathcal{A}_2} & \overset{def}{=} \{0, 1, 2\} \\
\$^{\mathcal{A}_2} & \overset{def}{=} \{(e_1, 2), (e_2, 1), (ERRC, 0)\} \\
undef_C^{\mathcal{A}_2} & \overset{def}{=} 0 \\
err_C^{\mathcal{A}_1} & \overset{def}{=} ERRC \\
s_1^{\mathcal{A}_2} & \overset{def}{=} 1 \\
s_2^{\mathcal{A}_2} & \overset{def}{=} 2\ ,
\end{array}$$

Fig. 15. A Σ_{EX_2}-algebra, named $\mathcal{A}_2$.



$$\begin{array}{rl}
C^{I_6} & \overset{def}{=} \{e_1, e_2\} \\
\$^{I_6} & \overset{def}{=} \{(e_1, undef_C), \\
 & \quad (e_2, s_2)\} \\
C\text{-}STATES^{I_6} & \overset{def}{=} \{s_1, s_2, undef_C\} \\
undef_C^{I_6} & \overset{def}{=} undef_C \\
s_1^{I_6} & \overset{def}{=} s_1 \\
s_2^{I_6} & \overset{def}{=} s_2
\end{array}$$

Fig. 16. (a) Instance diagram, I_6. (b) Algebraic denotation.

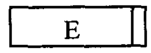
Here, a is an attribute of X that evaluates to an object of class Y. Formally, an attribute a, as above, is specified by including its signature,

$$a : X \to Y,$$

in the specification for class X. We add one constraint on attribute functions: any attribute function a maps error object to the error object of the other class. That is,

$$a(err_X) \overset{def}{=} err_Y.$$

External classes are, by definition, already specified algebraically; hence the formalization of an external class is essentially a matter of stating our expectations about how the specification of that class is defined. If an object model contains a class E given by a picture of the form



then it is assumed that there exists a trait named *CLASS-E* and that this trait specifies the sort E. It is up to the specifier to construct the specification *CLASS-E*. The primary purpose of external classes is to make use of the LSL trait handbook [22], but any trait can be incorporated in this way.

A more general type of external class is one that is parameterizable. Fig. 17 shows the general graphical form for a parameterizable class. This diagram, instead of specifying a class in itself, specifies how to construct a class. The external class X is assumed to already have a specification that gives E as its parameter. Since class Y is not external, its specification can be derived independently of the class X. If an association, an attribute function, or any other structure makes reference to this

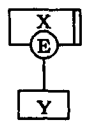


Fig. 17. General form for a parameterized class.

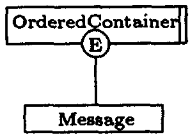


Fig. 18. A parameterizable class with a bound parameter.

```
CLASS-OrderedContainer(E) : trait
    includes
        OrderedContainer( OrderedContainer for C )
```

Fig. 19a. CLASS-OrderedContainer, specification of class encapsulating a parameterized trait.

```
OrderedContainer (E) : trait
    introduces
        new      : → C
        insert   : E,C → C
        isEmpty  : C → Bool
        head     : C → E
        tail     : C → C
    asserts
        C generated by new, insert
        C partitioned by head, tail, isEmpty
        ∀c : C, e : E
            isEmpty(new) == true
            not(isEmpty(insert(e,c))) == true
            head(insert(e,new)) == e
            tail(insert(e,new)) == new
```

Fig. 19b. OrderedContainer, a simple parameterized trait.

external class X, then the following inclusion rules are required in the **includes** section of its trait.

```
includes
    CLASS-Y,
    CLASS-X (Y for E)
```

As an example, consider the diagram given in Fig. 18. Here, the class Message has been bound to the E parameter of the OrderedContainer class (specified in Fig. 19a), which includes the OrderedContainer trait (specified in Fig. 19b). From the formal perspective, there are two levels of renamings necessary in order to obtain the proper parameterization. First, for sort C, we rename the sort name of the container from the OrderedContainer trait, in order for the new trait, CLASS-OrderedContainer to use the sort naming convention we have imposed. Second, the class Message has been bound to the E parameter of the OrderedContainer class; this binding corresponds to a renaming of the parameter E within the CLASS-OrderedContainer trait. Therefore, wherever CLASS-OrderedContainer is needed, traits must be included as follows:

```
includes
    CLASS-Message,
    CLASS-OrderedContainer (Message for E)
```

Fig. 20 specifies attributes *mail* and *addr* for a Mailbox object. The attribute *mail* represents a collection of Message objects, since its specification indicates that it results in an object of class OrderedContainer(Message). The attribute *addr* characterizes a Mailbox by giving its Address. The specification for the class Mailbox (shown in Fig. 21) is derived using the rules stated above.

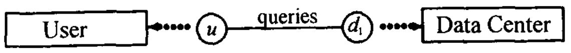
In this section, we have described the semantics for the simplest components of object model and instance diagrams, namely classes, objects, and attributes. Fig. 22 summarizes the object model semantics introduced in this section, and Fig. 23 summarizes the instance diagram semantics. Next, the semantics of associations are added to the formalization, including an introduction to organizational conventions for assembling specifications into modules.

B. Binary Associations

In this section we extend the class, object, and attribute semantics given in the previous section to include binary associations. Binary associations are recast as predicates, and techniques for constructing modular algebraic specifications directly from an object model are given.

B.1. General Binary Associations

Recall that the instance diagrams given in Figs. 2 and 4 depict two different instances of the *queries* association, the latter being inconsistent with the multiplicity constraints of object O_1, given in Fig. 1. In general, an instance of an n-ary association R consists of a set of *R-links*, where an R-link is an n-tuple of object names. For example, from Fig. 2, the subgraph



depicts a single *queries*-link, which can be formally represented by the 2-tuple (u,d_1). Hence, the instance of *queries* given in Fig. 4 can be characterized by the set of n-tuples

$$\{(u, d_1), (u, d_3)\}, \tag{1}$$

while the instance of *entryPoint* given in Fig. 4 can be characterized by the set of n-tuples

$$\{(n_1, u), (n_2, u)\}. \tag{2}$$

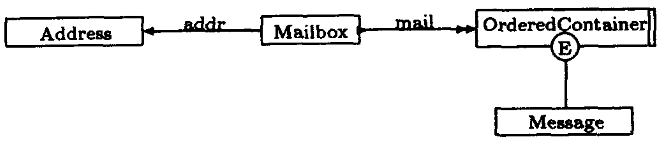


Fig. 20. An attribute *mail* whose valuation is an external class.

```
CLASS-Mailbox : trait
    includes
        CLASS-Message,
        CLASS-OrderedContainer ( Message for E ),
        CLASS-Address,
    introduces
        err_MB     : → Mailbox
        undef_MB   : → Mailbox-STATES
        $          : Mailbox → Mailbox-STATES
        mail       : Mailbox → OrderedContainer
        addr       : Mailbox → Address
    asserts
        Mailbox-STATES generated by undef_MB
        $(err_MB) = undef_MB
```

Fig. 21. The specification for the class Mailbox.

Definition (Semantics of object models I) :
Let $\mathcal{O}$ be an object model. The semantics of $\mathcal{O}$ is an algebraic specification satisfying the following data.

(OM1) **Each class C in the object model $\mathcal{O}$ is denoted by a sort of the same name.**

(OM2) **For each class C, a sort $C\text{-}STATES$ is introduced as well as two nullary functions given by**

$$undef_C : \ \rightarrow C\text{-}STATES \ , \ err_C : \ \rightarrow C \ .$$

(OM3) **Each object-state s, for which a double-headed arrow leads from a class C to the oval containing s, is denoted by a function with signature $s : \ \rightarrow C\text{-}STATES$, and for every pair of object-states s_1 and s_2, the axiom $s_1 \neq s_2$ is included.**

(OM4) **For each class C, a valuation function '\$' is introduced with the signature**

$$\$: C \rightarrow C\text{-}STATES \quad .$$

The valuation of the error object is added as an axiom:

$$\$(err_C) = undef_C \quad .$$

(OM5) **If there is a double-headed arrow labeled a (to indicate an attribute), leading from a class C to a class D (which depicts an attribute a of C), then the function signature**

$$a : C \rightarrow D \quad ,$$

is added to specification for class C.

(OM6) **If the class D in rule (OM5) is an external class, then the trait for D is included by the specification for C. If D has no parameters, then the clause**

```
includes
        CLASS-D
```

is added. If D has parameters $p_1, \ldots, p_k$, and there is a line connecting each p_i to a class q_i, then the following clause is added:

```
includes
        CLASS-p1, ..., CLASS-pk,
        CLASS-D ( q1 for p1, ..., qk for pk)
```

Fig. 22. The semantics of simple object models (no associations).

Sets of ordered n-tuples, such as those given in (1) and (2), are called *relations*. Rather than using this set-based representation of relations, we represent relations as predicates. For example, the relation given in (1) may be equivalently denoted by introducing a predicate named *queries* having the following semantics:

$$queries(u, d_1) \overset{def}{=} true$$

$$queries(u, d_2) \overset{def}{=} false$$

$$queries(u, d_3) \overset{def}{=} true$$

Similarly, the relation given in (2) is denoted by the following semantics for the *entryPoint* predicate:

$$entryPoint(n_1, u) \overset{def}{=} true \tag{3}$$

$$entryPoint(n_2, u) \overset{def}{=} true$$

Definitions (Semantics of instance diagrams I) :
Let $\mathcal{O}$ be an object model, and let $\mathcal{I}$ be an instance diagram. The semantics of $\mathcal{I}$ with respect to object model $\mathcal{O}$, is an algebra given (in part) by the following data.

(ID1) **For each class C in the object model $\mathcal{O}$, the carrier set $C^{\mathcal{I}}$ is the set of objects $\{x_1, \ldots, x_k\}$ given in $\mathcal{I}$ for which a dotted arrow leads from each x_i to C, and also includes an object named err_C.**

(ID2) **The carrier set $C\text{-}STATES^{\mathcal{I}}$ is the set of object-state names, including $undef_C$, given in $\mathcal{O}$ for which a double-headed arrow leads from C to the object-state.**

(ID3) **For each object e of class C in $\mathcal{I}$, if there is an unlabeled double-headed arrow leading from e to some object-state s, then**

$$\$^{\mathcal{I}}(e) \overset{def}{=} s \quad .$$

If there is no double-headed arrow from e to any object-state s, then

$$\$^{\mathcal{I}}(e) \overset{def}{=} undef_C \quad .$$

(ID4) **For each object e of class C in $\mathcal{I}$, if there is double-headed arrow labeled a leading from e to some object o, then**

$$a^{\mathcal{I}}(e) \overset{def}{=} o \quad .$$

(ID5) **For every attribute a given in object model $\mathcal{O}$, where**

$$a : C \rightarrow D \quad ,$$

if e is an object of C but no double-headed arrow, labeled by a, is shown leading from e to any object of D, then

$$a^{\mathcal{I}}(e) \overset{def}{=} err_D \quad .$$

(ID6) **For each object-state s, $s^{\mathcal{I}} \overset{def}{=} s$.**

(ID7) **For each class C,**

$$undef_C^{\mathcal{I}} \overset{def}{=} undef_C \quad ,$$
$$err_C^{\mathcal{I}} \overset{def}{=} err_C \quad .$$

Fig. 23. The semantics of simple instance diagrams (those having no associations).

Formally, an association is denoted by a predicate of the same name. Let O be an object model with classes $C = \{C_1, \ldots, C_n\}$. Let R be the name of an association in O, and let $D_1, \ldots, D_k$, where $D_i \in C$, for $1 \leq i \leq k$, be the classes associated by R. Then R is denoted by a predicate whose signature is given by

$$R : D_1, \ldots, D_k \rightarrow BOOL,$$

where the comma between the sort names represents the Cartesian product. For example, the predicate for the *queries* association has the signature:

$$queries : User, Data\ Center \rightarrow BOOL.$$

For each association, a set of constraints are added to account for the error objects on the predicate denoting the association. Formally, for each binary association R, we add the following axioms:

$$(\forall x : X, y : Y \ . \ R(err_X, y) \wedge x \neq err_X \Rightarrow \neg R(x, y))$$

$$(\forall x : X, y : Y \ . \ R(x, err_Y) \wedge y \neq err_Y \Rightarrow \neg R(x, y))$$

Informally, these two constraints ensure that any object related to an error object is related to no other object via the same association. The statement ($\forall x : X \,.\, P$) is read "for all elements x of sort X, the statement P evaluates to true," and a similar reading applies when the existential quantifier '$\exists$' is used in place of '$\forall$'.

Now, consider the object model O_1 given in Fig. 1. Given the formalization techniques presented thus far, it is possible to obtain several equivalent specifications for this diagram, which differ in the way they are modularized. As a convention, we extract the specifications corresponding to classes first, followed by the specifications for the associations, which are developed by importing the relevant class specifications. The specification for a class C is a trait named *CLASS-C*, and for an association R, the trait name *ASSOCIATION-R* is used. This naming convention for association specifications does not allow the overloading of an association name R. If an association name R is used to name several different associations, then associated class names are included as part of the name given to its specification. For the Data Center and User classes given in Fig. 1, the specifications given in Figs. 24 and 25 are derived, respectively. The specifications for the other two classes are similar.

Temporarily omitting the specification for multiplicity constraints on the associations, Fig. 26 gives a partial specification for the queries association, where the specifications for the other associations would have a similar form. This specification simply introduces the predicate *queries* and makes the standard assertions about the role of the error object (i.e. any object related to the error object is not related to any other object).

B.2. Multiplicity Constraints

Multiplicity constraints can be described in terms of four relational properties: *functional*, *injective*, *surjective*, and *total*. For convenience, we define the following second-order predicates, which will be converted into first-order predicates when they are incorporated into our algebraic specifications. Let R be a predicate (denoting an association) with signature

$$R : A, B \to BOOL.$$

```
CLASS-DataCenter : trait
        introduces
                undef_DC  →DataCenter-STATES
                err_DC  :  →DataCenter
                $       : DataCenter →DataCenter-STATES
        asserts
                DataCenter-STATES generated by undef_DC
                $(err_DC) = undef_DC
```

Fig. 24. Trait specifying the class DataCenter.

```
CLASS-User : trait
        introduces
                undef_User   :  →User-STATES
                err_User     :  →User
                $            : User →User-STATES
        asserts
                User-STATES generated by undef_User
                $(err_User) = undef_User
```

Fig. 25. Trait specification for class User.

R is *functional* from A to B if every object of A, with the possible exception of the error object, is related to at most one object of B:

functional(R, A, B)

$$\overset{def}{=} \big(\forall a : A, x, y : B \,.\, \big(a \neq err_A \wedge R(a,x) \wedge R(a,y) \Rightarrow x = y\big)\big).$$

Note that the error object exclusion is necessary to ensure that the semantics of error objects is consistent. R is *injective* from A to B if every object of B, with the possible exception of the error object, is related to at most one object of A:

injective(R, A, B)

$$\overset{def}{=} \big(\forall x, y : A, b : B \,.\, \big(b \neq err_B \wedge R(x,b) \wedge R(y,b) \Rightarrow x = y\big)\big).$$

A relation R is *surjective* from A to B if every element of B is related to some element of A:

$$\textbf{surjective}(R, A, B) \overset{def}{=} \big(\forall b : B \,.\, (\exists a : A \,.\, R(a,b))\big).$$

R is *total* from A to B if every element of A is related to some element of B, or, formally:

$$\textbf{total}(R, A, B) \overset{def}{=} \big(\forall a : A \,.\, (\exists b : B \,.\, R(a,b))\big).$$

For the last two predicates, **total** and **surjective**, the error object exception is not required. This omission is allowed because we can always associate the error objects of different classes with each other.

Given any binary association R over classes A and B, the multiplicity constraints of R can be completely characterized by a conjunction of zero or more instances of the relational predicates given above. Fig. 27 uses the four relational predicates to specify the multiplicity constraints for seven different combinations of endpoints. The column titled "Object Model" gives an object model containing a single association. The column titled "Example" provides an instance diagram depicting a state that is consistent with the corresponding object model; each circle represents a distinct object, though we have omitted the names of the objects for simplicity. The rightmost column, titled *Constraints(R,A,B)*, contains the specification for the multiplicity constraints of the association R as given in the first column. Since the same association names are used in different examples throughout this paper, we subscript the *Constraints* function by the name of the object model that is being referenced. For example,

$$Constraints_O(R,A,B)$$

refers to the specification of an association R as given in some object model O. The example instance diagrams given

```
ASSOCIATION-queries : trait
        includes
                CLASS-User, CLASS-DataCenter
        introduces
                queries: User, DataCenter → BOOL
        asserts
        ∀u :User, d:DataCenter
                queries(err_User, d) ∧ u ≠ err_DC ⇒ ¬queries(u, d)
                queries(u, err_DC) ∧ u ≠ err_User ⇒ ¬queries(u, d)
```

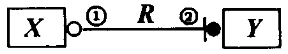
Fig. 26. Partial specification of the *queries* association.

in Figs. 27a–27g were chosen carefully; for any pair of object models O and $\mathcal{T}$ in Fig. 27, if

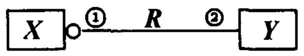
$$\neg(Constraints_O(R, A, B) \Rightarrow Constraints_{\mathcal{T}}(R, A, B)),$$

then the example instance diagram given for object model O is inconsistent with object model $\mathcal{T}$.

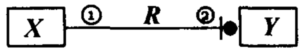
The object models shown in Fig. 27, henceforth referred to collectively as the *basis object models*, represent only seven of the fifteen possible combinations of multiplicity constraints.[3] For example, there is no basis object model showing the specification for the combination of multiplicity constraints given in the object model O_4 below.



However, it is straightforward to derive the specification for these multiplicity constraints by using the basis object models in the following manner. We decompose the above object model into the two underlying basis object models:



(4)



(5)

Let $\mathcal{P}_1$ refer to the object model given in (4) and $\mathcal{P}_2$ refer to the object model given in (5). The specifications for object models $\mathcal{P}_1$ and $\mathcal{P}_2$ can be obtained directly from the basis object models (c) and (f), respectively, yielding

$$Constraint_{\mathcal{P}_1}(R, X, Y) = \textbf{injective}(R, X, Y) \wedge \textbf{total}(R, X, Y) \wedge \textbf{functional}(R, X, Y)$$

and

$$Constraint_{\mathcal{P}_2}(R, X, Y) = \textbf{total}(R, X, Y) \wedge \textbf{surjective}(R, X, Y) \wedge \textbf{injective}(R, X, Y).$$

The decomposition of O_4 into $\mathcal{P}_1$ and $\mathcal{P}_2$ illustrates a special property of which we will now take advantage. Object model $\mathcal{P}_1$ has a more constraining endpoint on the Y-end of association R than does O_4, and the same is true for the X-end of association R in $\mathcal{P}_2$. Hence, if we compose $\mathcal{P}_1$ and $\mathcal{P}_2$. in such a way that from the two corresponding endpoints, the least constraining endpoint is always selected, then we will once again have the object model O_4. Mathematically, we want to construct the most constraining specification possible that is no stronger than either of the specifications given by $\mathcal{P}_1$ or $\mathcal{P}_2$. A simple method for computing this desired specification is to let $Constraint_{O_4}(R, X, Y)$ be the conjunctive expression whose conjuncts comprise those conjuncts common to both $Constraint_{\mathcal{P}_1}(R, X, Y)$ and $Constraint_{\mathcal{P}_2}(R, X, Y)$. That is, we obtain the following multiplicity constraint for association R of object model O_4:

$$Constraint_{O_4}(R, X, Y) = \textbf{injective}(R, X, Y) \wedge \textbf{total}(R, X, Y).$$

3. Actually, Figs. 27a, b, d, and f constitute the basis object models; the other three are redundant. However, it is easier to provide these three redundant object models for quick reference than to describe a calculation by which their specifications can be derived.

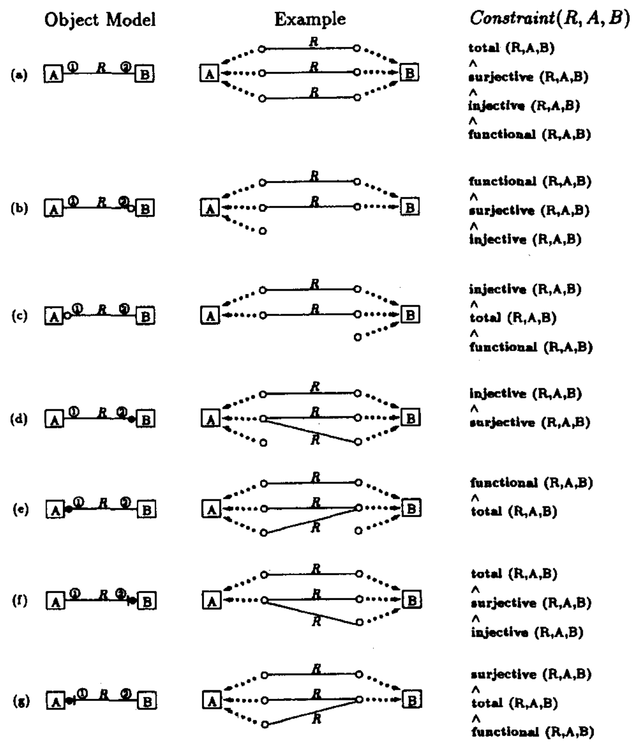

Object Model
Example
Constraint(R, A, B)
(a)
A ① R ② B
total (R,A,B) ∧ surjective (R,A,B) ∧ injective (R,A,B) ∧ functional (R,A,B)
(b)
functional (R,A,B) ∧ surjective (R,A,B) ∧ injective (R,A,B)
(c)
injective (R,A,B) ∧ total (R,A,B) ∧ functional (R,A,B)
(d)
injective (R,A,B) ∧ surjective (R,A,B)
(e)
functional (R,A,B) ∧ total (R,A,B)
(f)
total (R,A,B) ∧ surjective (R,A,B) ∧ injective (R,A,B)
(g)
surjective (R,A,B) ∧ total (R,A,B) ∧ functional (R,A,B)
R

Fig. 27. The basis object model and their specifications.

Therefore, given any binary association with any combination of multiplicity constraints, it is possible to express these constraints using the four relational predicates, and the required specification can be derived from the seven basis object models given in Fig. 27.

The specifications given above for multiplicity constraints are not appropriate for incorporation into algebraic specifications. For example, only first-order axioms are allowed in an algebraic specification. Due to the simplicity of our relational predicates, it is easy to obtain a first-order characterization of any resulting specifications by treating the predicate definitions as *parameterized specifications*. A reference to any of the four relational predicates, for example

$$\textbf{surjective}(\textit{queries, User, DataCenter}), \qquad (6)$$

is replaced with the corresponding predicate definition, where the actual parameters given in the reference replace the corresponding formal parameters given in the predicate's definition. We refer to this technique as *unfolding*. Unfolding predicate (6) yields the first-order assertion

$$\big(\forall d : DataCenter\ .\ \big(\exists u : User\ .\ queries(u, d)\big)\big). \qquad (7)$$

While of the first-order, this latter assertion is still not admissible as an axiom. Most algebraic specification languages require that all axioms be in a form known as the universal closure [16]. The universal closure of an axiom has the form

$$\big(\forall x_1 : X_1, x_2 : X_2, \ldots, x_n : X_n\ .\ E\big),$$

where E is an expression containing no quantifiers, and where every variable in E is given as one of the variables $x_1, \ldots, x_n$.

Since existential quantifiers are not allowed, we eliminate them using the standard technique of *skolemization* [26].

Skolemization of a formula is the elimination of existential quantifiers and their bound variables by replacing the variables with a *Skolem function.* A "skolemized" equivalent of (7) above is

$$(\forall d : \mathit{DataCenter} \,.\, \mathit{queries}(\mathit{skolem}(d), d)),$$

where the signature of the Skolem function *skolem* is given by

$$\mathit{skolem} : \mathit{DataCenter} \rightarrow \mathit{User}.$$

The techniques of unfolding and skolemization provide the basic mathematical tools needed to include multiplicity constraints in algebraic specifications.

The specification for the *queries* association depicted in Fig. 1 is obtained by intersecting the conjuncts from the basis object models in Figs. 27c and 27d. The resulting specification of queries with respect to object model O_1 is given by

$$\mathit{Constraint}_{O_1}(\mathit{queries}, \mathit{User}, \mathit{DataCenter}) =$$
$$\mathbf{injective}(\mathit{queries}, \mathit{User}, \mathit{DataCenter}).$$

We transform this specification into axiom form by unfolding each of the conjuncts and skolemizing where necessary. This process yields the following formula:

$$(\forall u_1, u_2 : \mathit{User}, d_1 : \mathit{DataCenter} \,.\, (\mathit{queries}(u_1, d_1) \wedge \mathit{queries}(u_2, d_1)) \Rightarrow u_1 = u_2).$$

Combining this formula with the partial algebraic specification for *queries*, given in Fig. 26, completes the specification. The resulting specification is given in Fig. 28.

In order to demonstrate how the formalization enables a rigorous determination of inconsistencies between an object model and an instance diagram, we consider again the relationship between the object model O_1 given in Fig. 1 and the instance diagram I_1 given in Fig. 2. We claimed earlier that I_1 is consistent with O_1. Since the specification *ASSOCIATION-queries*, given above, was derived from O_1, we can now formally conclude that the algebra corresponding to I_1 is consistent with the specification *ASSOCIATION-queries*. The algebra corresponding to instance diagram I_1 (with respect to object model O_1) is given in Fig. 29.

B.3. Aggregation Associations

Aggregation is the composition of parts into a whole, and an aggregation association is used to specify the object parts that contribute to a whole object. The notation and semantics of aggregation is based on a binary association: a whole object is related to each of its parts. However, aggregation is differentiated from general binary associations in the way that it is used.

Consider, for example, the object model pictured in Fig. 30. The Ping class is a subtype of the class RequestMsg, which was given in Fig. 9; RequestMsg, in turn, is a subtype of the class Message, given in Fig. 8. Because of these subtype relationships, the Ping class "inherits" the aggregation associations of each of its supertypes, and the "flattened" object model for Ping is shown in Fig. 30. Of particular note are the parts *replyAddress* and *sender*, both of which refer to objects from the class Address. If we were to consider aggregation to be a notational variation on the general binary association, then the relationship between Ping messages and Addresses would be one-to-one. The object model pictured in Fig. 30 clearly sug-

```
ASSOCIATION-queries : trait
        includes
                CLASS-User, CLASS-DataCenter
        introduces
                queries: User, DataCenter → BOOL
        asserts
                $\forall u_1, u_2 : User,\ d_1 : DataCenter$
                $queries(err_{User}, d) \wedge u \neq err_{DC} \Rightarrow \neg queries(u, d)$
                $queries(u, err_{DC}) \wedge u \neq err_{User} \Rightarrow \neg queries(u, d)$
                $(queries(u_1, d_1) \wedge queries(u_2, d_1)) \Rightarrow u_1 = u_2$
```

Fig. 28. Completed specification for the *queries* association.

$User^{\mathcal{I}_1}$	$\stackrel{def}{=}$	$\{u, ERRUSER\}$
$NetworkManager^{\mathcal{I}_1}$	$\stackrel{def}{=}$	$\{ERRNETMAN\}$
$Dataset^{\mathcal{I}_1}$	$\stackrel{def}{=}$	$\{u, ERRDATA\}$
$DataCenter^{\mathcal{I}_1}$	$\stackrel{def}{=}$	$\{d_1, d_2, d_3, ERRDC\}$
$User\text{-}STATES^{\mathcal{I}_1}$	$\stackrel{def}{=}$	$\{undef_{User}\}$
$NetworkManager\text{-}STATES^{\mathcal{I}_1}$	$\stackrel{def}{=}$	$\{undef_{NM}\}$
$Dataset\text{-}STATES^{\mathcal{I}_1}$	$\stackrel{def}{=}$	$\{undef_{Dataset}\}$
$DataCenter\text{-}STATES^{\mathcal{I}_1}$	$\stackrel{def}{=}$	$\{undef_{DC}\}$
$undef^{\mathcal{I}_1}_{User}$	$\stackrel{def}{=}$	$undef_{User}$
$undef^{\mathcal{I}_1}_{DC}$	$\stackrel{def}{=}$	$undef_{DC}$
$undef^{\mathcal{I}_1}_{NM}$	$\stackrel{def}{=}$	$undef_{NM}$
$undef^{\mathcal{I}_1}_{Dataset}$	$\stackrel{def}{=}$	$undef_{Dataset}$
$err^{\mathcal{I}_1}_{User}$	$\stackrel{def}{=}$	$ERRUSER$
$err^{\mathcal{I}_1}_{NM}$	$\stackrel{def}{=}$	$ERRNETMAN$
$err^{\mathcal{I}_1}_{Dataset}$	$\stackrel{def}{=}$	$ERRDATA$
$err^{\mathcal{I}_1}_{DC}$	$\stackrel{def}{=}$	$ERRDC$
$\$^{\mathcal{I}_1}_{User}$	$\stackrel{def}{=}$	$\{(u, undef_{User}), (err_{User}, undef_{User})\}$
$\$^{\mathcal{I}_1}_{NM}$	$\stackrel{def}{=}$	$\{(err_{NM}, undef_{NM})\}$
$\$^{\mathcal{I}_1}_{Dataset}$	$\stackrel{def}{=}$	$\{(err_{Dataset}, undef_{Dataset})\}$
$\$^{\mathcal{I}_1}_{DC}$	$\stackrel{def}{=}$	$\{(d_1, undef_{DC}), (d_2, undef_{DC}), (d_3, undef_{DC}), (err_{DC}, undef_{DC})\}$
$queries^{\mathcal{I}_1}(u, d_1)$	$\stackrel{def}{=}$	$true$
$queries^{\mathcal{I}_1}(u, d_3)$	$\stackrel{def}{=}$	$true$
$queries^{\mathcal{I}_1}(err_{User}, d_2)$	$\stackrel{def}{=}$	$true$

Fig. 29. An algebra corresponding to instance diagram I_1.

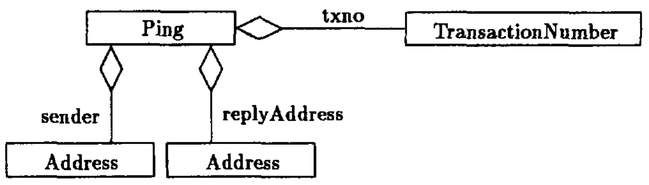


Fig. 30. The Ping message-type, with its inherited parts shown.

gests that there are two Address objects associated with a Ping message: one corresponding to the sender of the message, and the other corresponding to the destination for responses to the message. A semantics for aggregation must take into account that the endpoints of the aggregation association are treated somewhat differently than those of general binary associations.

An aggregation association between a class X and a class Y, where X is the aggregate, is formally specified as a predicate named *hasPart* with signature given as follows:

$$hasPart : X, Y \rightarrow BOOL.$$

The first argument for *hasPart* is the aggregate and the second argument is always a part. However, the endpoints of the aggregation association cannot be interpreted, directly, as contraints on the association. Instead, endpoints are interpreted as constraints on the part name given as a label. For example, in Fig. 30 the endpoint at the Address-end of the association labeled *sender* is the "exactly one" endpoint. We interpret this notation to mean that "there is exactly one *sender* Address that is a part of any given Ping message." Part names are thus mappings from an aggregate object to its parts.

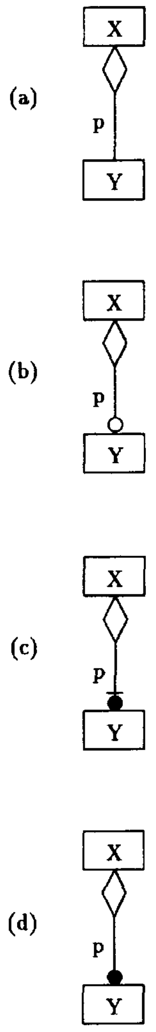
In order to complete the semantics of aggregation, we now consider how part name mappings are specified and how they constrain the predicate *hasPart* for any arbitrary aggregation. Fig. 31 depicts the four possible endpoints for the part-end of an aggregation association with a part named *p*. For each situation, the signature of the part name *p* is given, and the set of constraints on *p* and *hasPart*, induced by the given endpoint, are stated. For constraints that require the inclusion of other specifications, the appropriate details are given.

In Figs. 31a and 31b, an object of class X can have at most one Y-part named p. Hence, when formalizing p as a mapping, p can be modeled as a function from X to Y, as shown by the signature of p given for these two figures. The difference between these two figures is that an object model using the form given in Fig. 31a must always have a p-part for every object x. However, the error object is always used to explicitly indicate the absence of a relationship. Therefore, the required constraint

$$(\forall x : X \,.\, p(x) \neq err_Y)$$

asserts that the p-part of any object x is never the error object, and hence some regular object. This last constraint cannot be required for the object model given in Fig. 31b, since this model asserts that a p-part is optional.

In Figs. 31c and 31d, an object of class X can have many Y-parts named p. Hence, when formalizing p as a mapping, p *cannot* be modeled as a function from X to Y. Instead, p is modeled as a mapping of objects from class X to *sets* of objects from class Y. LSL has no predefined notation for specify-



(a)
introduces
$p : X \rightarrow Y$
asserts
$(\forall x : X \,.\, p(x) \neq err_Y)$

(b)
introduces
$p : X \rightarrow Y$

(c)
includes
Set(*Y-SET* **for** *C*, *Y* **for** *E*)
introduces
$p : X \rightarrow Y\text{-}SET$
asserts
$(\forall x : X \,.\, p(x) \neq \{\})$

(d)
includes
Set(*Y-SET* **for** *C*, *Y* **for** *E*)
introduces
$p : X \rightarrow Y\text{-}SET$

Fig. 31. The four part-end endpoint configurations for aggregation.

ing power-sets, but we can make use of an existing specification of power-sets from the LSL Trait Handbook [22], the *Sets* trait. The trait *Sets*, given in Appendix I, is a simple axiomatization of elementary set theory. It specifies set membership, intersection, union, difference, and other set operations. The two sorts used by this specification are the sorts E, denoting the element type, and C, denoting the set type. In Figs. 31c and 31d, it is indicated that this trait be `included` with its parameters E and C renamed to Y and Y-SET, respectively. By doing so, the signature of p can be given as

$$p : X \rightarrow Y\text{-}SET.$$

The constraint

$$(\forall x : X \,.\, p(x) \neq \{\})$$

asserts that every x must have some p-objects, which is what the object model given in Fig. 31c asserts.

This indirect route to deriving specifications is needed only because a specifier might give names to different parts of the same class, a situation illustrated earlier by Fig. 30. The formalization presented thus far has considered each part separately. The remaining constraints on aggregation consider the relationships between all parts of the same type.

In order to simplify the introduction of constraints, we introduce an auxilliary predicate, *p-part*, for each part name *p*. As above, suppose that *p* is the part name label on an aggregation association between classes *X* and *Y*, with *X* the aggregate; then

$$p\text{–}part : X, Y \rightarrow BOOL.$$

If *p* is a part name of the type shown in Fig. 31a or 31b, then

$$p-part(x,y) \overset{\text{def}}{=} p(x) = y$$

If *p* is a part name of the type shown in Fig. 31c or 31d, then

$$p-part(x,y) \overset{\text{def}}{=} y \in p(x)$$

Now, suppose that a class *X* has *k* part names $p_1, \ldots, p_k$, all of which map into the same class, *Y*. Then for each pair of part names p_i and p_j $(i \neq j)$, the following constraint must hold:

$$(\forall x : X, y : Y \,.\, \neg(p_i\text{–}part(x, y) \wedge p_j\text{–}part(x, y))).$$

That is, different part names are given to parts of the same class to differentiate them from one another; hence we require that these part name mappings produce different objects. Finally, we expect that the *hasPart* association is completely specified by its different part namings:

$$(\forall x : X, y : Y \,.\, hasPart(x, y) = p_1\text{–}part(x, y) \vee \quad \vee\, p_k\text{–}part(x, y)).$$

These rules can be applied to the aggregation association depicted in Fig. 30 to derive its specification. This figure considers only the aggregation associations between the class Ping and the class Address.

In this section, we have described the semantics of general binary associations, and the more specialized aggregation association. Fig. 33 summarizes the rules that have been introduced to define the semantics of binary associations. Fig. 34 summarizes the same for instance diagrams.

C. N-ary Association

Rumbaugh et al. [10] give a brief description of associations having arity greater than two but do not provide explicit description of the semantics of these higher-arity associations. In our analysis of the OMT notations for ternary and higher-arity associations, we establish an approach to semantics that is a generalization of that used for binary associations.

```
ASSOCIATION-hasPart-Ping-Address : trait
  includes
    CLASS-Ping,
    CLASS-Address
  introduces
    hasPart             : Ping, Address → BOOL
    sender              : Ping → Address
    replyAddress        : Ping → Address
    sender-part         : Ping, Address → BOOL
    replyAddress-part   : Ping, Address → BOOL
  asserts
    ∀p : Ping,  a : Address
        sender(p) ≠ err_Address
        replyAddress(p)  ≠  err_Address
        sender-part(p, a)  ==  sender(p) = a
        replyAddress-part(p, a)  ==
           replyAddress(p) = a
        ¬(sender-part(p, a)∧replyAddress-part(p, a))
        hasPart(p, a)  ==  sender-part(p, a)∨
           replyAddress-part(p, a)
```

Fig. 32. Specification for *sender* and *replyAddress* parts of Ping.

Definition (Semantics of Object Models II) :
Let $\mathcal{O}$ be an object model, and let R be a binary association in $\mathcal{O}$ relating objects from classes D_1 and D_2. $\mathcal{O}$ is denoted by an algebraic specification determined by the rules (OM1)-(OM6), given in Figure 22, in addition to the rules given below.

(OM7) **Association R is denoted by the predicate**

$$R : D_1, D_k \rightarrow BOOL \quad .$$

(OM8) **The endpoints of association R determine a set of axioms. Suppose the D_1-endpoint depicts a multiplicity of m and the D_2-endpoint depicts a multiplicity of n. Then the axioms are derived by the following steps:**

1. **decompose the m-to-n association R into an m-to-1 and 1-to-n binary association,**
2. **determine the second-order specifications, P_1 and P_2, of each of these associations using the basis schemata,**
3. **calculate the "intersection", P, of the specifications P_1 and P_2,**
4. **unfold and skolemize P, yielding a set of first-order axioms that are included in the trait for R.**

(OM9) **Error object constraints are introduced:**

$$\begin{aligned} &(\forall d_1 : D_1, d_2 : D_2 \;\bullet\; R(err_{D_1}, d_2) \wedge \\ &\qquad d_1 \neq err_{D_1} \Rightarrow \neg R(d_1, d_2)) \quad , \\ &(\forall d_1 : D_1, d_2 : D_2 \;\bullet\; R(d_1, err_{D_2}) \wedge \\ &\qquad d_2 \neq err_{D_2} \Rightarrow \neg R(d_1, d_2)) \quad . \end{aligned}$$

Fig. 33. The semantics of binary associations.

Definition (Semantics of Instance Diagrams II) :
Let $\mathcal{O}$ be an object model, and let R be a k-ary association in $\mathcal{O}$ relating objects from classes $D_1, \ldots, D_k$. Let $\mathcal{I}$ be an instance diagram. The denotation of instance diagram $\mathcal{I}$, with respect to object model $\mathcal{O}$, is given by an algebra. This algebra is determined by the rules (ID1)-(ID7), presented in Figure 23, and the following data.

(ID8) **For each R-link in $\mathcal{I}$, relating objects $d_1 : D_1, \ldots, d_k : D_k$,**

$$R^{\mathcal{I}}(d_1, \ldots, d_k) \overset{def}{=} true \quad .$$

(ID9) **For each n-tuple of objects $d_1 : D_1, \ldots, d_k : D_k$, if $\mathcal{I}$ has no R-link relating these objects, then:**

$$R^{\mathcal{I}}(d_1, \ldots, d_k) \overset{def}{=} false \quad .$$

Fig. 34. The semantics instance diagrams having binary associations.

As with binary associations, an *n*-ary association is specified as an *n*-ary predicate, but each edge of the association is considered separately when specifying multiplicity constraints. Each edge in an *n*-ary association is read as a binary association relating objects of one class to *n*-tuples of objects of the remaining classes, thus yielding *n* binary associations. The mathematical details that are necessary to rigorously describe the calculation of specifications for *n*-ary associations are illustrative of the method involved.

In order to illustrate our approach, consider the ternary association given in Fig. 35. This association is denoted by the predicate

$$accesses : User, DataCenter, Dataset \rightarrow Bool. \tag{8}$$

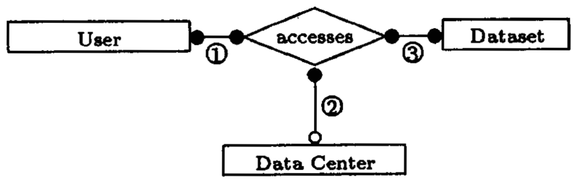


Fig. 35. A simple ternary association.

The order in which the three classes are given in this signature coincide with the numbering scheme given in Fig. 35. The *accesses* predicate must be sufficiently constrained to satisfy the multiplicity constraints given by the endpoints on each edge of the association.

The specification of multiplicity constraints is based on a decomposition of the association into three binary associations, graphically depicted in Fig. 36. As shown in this figure, the notation for ternary and higher-arity associations embeds many binary associations, each of which specifies a portion of the higher-arity association. In this figure, each edge of the ternary *accesses* association is "expanded" into a binary association. As a convention, we will refer to the edge that connects a class C to an association name R as the "C-edge of R." The User-edge of *accesses*, as given in Fig. 35, is shown as a binary association in object model $\mathcal{T}_1$ of Fig. 36, and a similar correspondence is seen for the DataCenter- and Dataset-edges in object models $\mathcal{T}_2$ and $\mathcal{T}_3$, respectively. The notation

$$\boxed{\text{DataCenter} \times \text{DataSet}}$$

depicts a class defined as the Cartesian product of the two classes DataCenter and Dataset:

In order to decompose the ternary association *accesses* into binary associations, these Cartesian product classes depicted in Fig. 36 must be represented as new sorts. For example, the class named *DataCenter* × *Dataset* given in object model $\mathcal{T}_1$ (Fig. 36) is denoted by a sort named *DataCenter-x-Dataset.* The substring "*-x-*" in the sort name is simply a mnemonic that makes it easier to remember what the sort name represents. We ensure that this sort does indeed represent the Cartesian product of the two named classes by introducing an ordered pair operator:

$$(\_,\_) : \mathit{DataCenter}, \mathit{Data} \rightarrow \mathit{DataCenter\text{-}x\text{-}Data}, \qquad (9)$$

where the symbol '__' in the function name is a marker that indicates the syntactical position of each argument (an *infix* syntax). For example, this signature asserts that given objects c and d of classes DataCenter and Dataset, respectively, the 2-tuple (c, d) is an object of sort *DataCenter-x-Dataset.* For each of the Cartesian product classes given in Fig. 36, we introduce a signature such as that given in (9).

$$(\_,\_) : \mathit{User}, \mathit{DataCenter} \rightarrow \mathit{User\text{-}x\text{-}DataCenter}$$

$$(\_,\_) : \mathit{User}, \mathit{Data} \rightarrow \mathit{User\text{-}x\text{-}Data}$$

Our formalization of classes does not require that the concept of object equality be explicitly defined. It was the basic premise of this formalization that every object has its own identity, and this identity is denoted by a handle that names the object. Cartesian product classes are artificial classes, the objects of which are n-tuples of objects from their component classes. It is therefore necessary to define what is meant by equality between ordered pairs. For each Cartesian product class *A-x-B*, we provide the axiom:

$$\left(\forall a_1, a_2 : A, b_1, b_2 : B \,.\, (a_1, b_1) = (a_2, b_2) \overset{\text{def}}{=} (a_1 = a_2) \wedge (b_1 = b_2)\right).$$

Hence, for the classes given above, we add the axioms:

$$\left(\forall u_1, u_2 : \text{User}, c_1, c_2 : \text{DataCenter} \,.\, (u_1, c_1) = (u_2, c_2)\right.$$
$$\left.\overset{\text{def}}{=} (u_1 = u_2) \wedge (c_1 = c_2)\right),$$

$$\left(\forall u_1, u_2 : \text{User}, d_1, d_2 : \text{Dataset} \,.\, (u_1, d_1) = (u_2, d_2)\right.$$
$$\left.\overset{\text{def}}{=} (u_1 = u_2) \wedge (d_1 = d_2)\right),$$

$$\left(\forall c_1, c_2 : \text{DataCenter}, d_1, d_2 : \text{Dataset} \,.\, (c_1, d_1) = (c_2, d_2)\right.$$
$$\left.\overset{\text{def}}{=} (c_1 = c_2) \wedge (d_1 = d_2)\right).$$

As with all class specifications, an error object must be introduced for each of the Cartesian product classes. The signatures for error objects of the three Cartesian product classes given above are as follows:

$$err_{\text{UxC}} : \rightarrow \mathit{User} - x - \mathit{DataCenter}$$
$$err_{\text{UxD}} : \rightarrow \mathit{User} - x - \mathit{Dataset}$$
$$err_{\text{CxD}} : \rightarrow \mathit{DataCenter} - x - \mathit{Dataset}$$

Each of the *accesses* associations given in Fig. 36 are different, and each is distinct from the ternary *accesses* association given in Fig. 35 from which they were derived. It is thus necessary to introduce a signature for each of these associations. The signature for *accesses* from object models $\mathcal{T}_1$, $\mathcal{T}_2$, and $\mathcal{T}_3$ are, respectively,

$$\mathit{accesses} : \mathit{User}, \mathit{DataCenter\text{-}x\text{-}Dataset} \rightarrow \mathit{BOOL}$$
$$\mathit{accesses} : \mathit{DataCenter}, \mathit{User\text{-}x\text{-}Dataset} \rightarrow \mathit{BOOL}$$
$$\mathit{accesses} : \mathit{Dataset}, \mathit{User\text{-}x\text{-}DataCenter} \rightarrow \mathit{BOOL}.$$

The relationship between these latter three predicates and the ternary accesses predicate given in (8) is described by the following three equations. Let $u : \mathit{User}$, $c : \mathit{DataCenter}$, and $d : \mathit{Dataset}$. Then we assert

$$\mathit{accesses}(u, (c, d)) = \mathit{accesses}(u, c, d), \qquad (10)$$

$$\mathit{accesses}(c, (u, d)) = \mathit{accesses}(u, c, d), \qquad (11)$$

$$\mathit{accesses}(d, (u, c)) = \mathit{accesses}(u, c, d). \qquad (12)$$

Similarly, the error objects of the Cartesian product classes are related to the error objects of the associated component classes.

$$err_{\text{UxC}} = (err_{\text{User}}, err_{\text{DC}})$$
$$err_{\text{UxD}} = (err_{\text{User}}, err_{\text{Data}})$$
$$err_{\text{CxD}} = (err_{\text{DC}}, err_{\text{Data}})$$

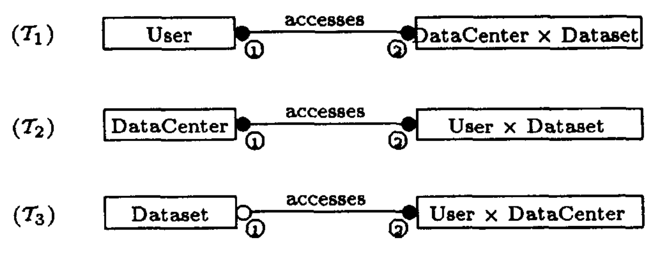


Fig. 36. The ternary *accesses* association, decomposed.

We are now able to demonstrate the calculation of the multiplicity constraints for the ternary *accesses* association.

Since the associations given in Fig. 36 are binary associations, the specifications for their multiplicity constraints are obtained in the manner indicated by the derivation rule OM8 (see Fig. 33). For each of these binary associations, the resulting constraints are given below. The multiplicity constraints of *accesses* given in object model $\mathcal{T}_1$ have the specification

$$Constraint_{\mathcal{T}_1}(accesses, User, DataCenter - x - Dataset) = true. \tag{13}$$

Since $\mathcal{T}_1$ depicts a many-to-many association, it is thus a simple relation. It has no multiplicity constraints. Similarly, with respect to object model $\mathcal{T}_2$, the specification of *accesses* is

$$Constraint_{\mathcal{T}_2}(accesses, Dataset, User - x - DataCenter) = true. \tag{14}$$

However, with respect to object model $\mathcal{T}_3$, we obtain

$$Constraint_{\mathcal{T}_3}(accesses, DataCenter, User - x - Dataset) = \mathbf{injective}(accesses, DataCenter, User - x - Dataset) \tag{15}$$

Each of the specifications given in (13), (14), and (15) give constraints that must be satisfied by the ternary *accesses* association. Hence the constraints of the ternary *accesses* consists of the conjunction of specifications given by (13), (14), and (15). Skolemization and unfolding are used to transform the specifications into axioms. For example, by applying unfolding and skolemization to the constraint given in (15), the following axiom is obtained:

$$\Big(\forall c_1, c_2 : DataCenter, u : User, d : Dataset\,.\; \big(\big(accesses(c_1,(u,d)) \wedge accesses(c_2,(u,d)) \wedge (u,d) \neq err_{\mathrm{UxD}}\big) \Rightarrow c_1 = c_2\big)\Big). \tag{16}$$

Note, however, that this axiom makes reference to the binary *accesses* predicates. By applying (10), (11), and (12), it is possible to replace all references to the binary *accesses* predicates with references to the ternary *accesses* predicate given in (8), thereby simplifying the overall specification. Applying these equations to the formula given in (16) yields:

$$\Big(\forall c_1, c_2 : DataCenter, u : User, d : Dataset\,.\; \big(accesses(c_1,u,d) \wedge accesses(c_2,u,d) \wedge u \neq err_{\mathrm{User}} \wedge d \neq err_{\mathrm{Dataset}}\big) \Rightarrow c_1 = c_2\Big).$$

The complete specification for the schema given in Fig. 35 can now be presented. Following the conventions for specification construction described in this paper, we derive the specifications for the object model classes first. The classes User and DataCenter were specified in Fig. 25 and Fig. 24. The remaining class of this ternary association, Dataset, is specified in Fig. 37.

Finally, we derive the specification for the ternary *accesses* association. The signature portion of the specification is easily obtained, and the axioms of the specification are those formulas that result from transforming the predicates

$$Constraint_{\mathcal{T}_1}(accesses, User, DataCenter - x - Dataset),$$
$$Constraint_{\mathcal{T}_2}(accesses, Dataset, User - x - DataCenter),$$
$$Constraint_{\mathcal{T}_3}(accesses, DataCenter, User - x - Dataset).$$

according to the method described in this section. The result is given in Fig. 38.

```
CLASS-Dataset : trait
    introduces
        undef_Dataset  :  →Dataset-STATES
        err_Dataset    :  →Dataset
        $              : Dataset →Data-STATES
    asserts
        Dataset-STATES generated by undef_Dataset
        $(err_Dataset) = undef_Dataset
```

Fig. 37. Trait specification for the class Dataset.

```
ASSOCIATION-accesses : trait
includes
        CLASS-User,
        CLASS-DataCenter,
        CLASS-Dataset
introduces
        accesses      : User, DataCenter, Dataset → BOOL
asserts
        ∀u : User, c1, c2 : DataCenter, d : Dataset
        % Multiplicity constraints for User-edge
              true
        % Multiplicity constraints for Dataset-edge
              true
        % Multiplicity constraints for DataCenter-edge
        accesses(u, c1, d) ∧ accesses(u, c2, d) ∧ (u ≠ err_User)∧
          (d ≠ err_Dataset) ⇒ c1 = c2
```

Fig. 38. The specification for the ternary association *accesses*.

D. Subtyping

Subtyping is a relationship between classes, but it is unlike the associations used in OMT. Subtyping creates a taxonomy for classifying objects. The guiding principle in our formalization of subtyping is summarized as follows:

if A is a subtype of B, then objects of type A are also objects of type B.

If *A* is a subtype of *B*, then it is expected that any object of class *A* can be treated as if it were an object of class *B*. Hence, in order to introduce subtypes, we must consider exactly what is allowed to be observed about a class of objects. This concept is the fundamental property that our formalization of subtyping will capture.

Let *X* and *Y* be classes. Suppose it has been asserted that *X* is a subtype of *Y*, as depicted in Fig. 39. We say that *Y* is a *nominal* subtype of *X* (the specifier has named *Y* as a subtype of *X*). Then we introduce a mapping from *X* to *Y*:

$$simulates : X \rightarrow Y.$$

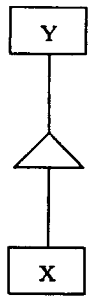


Fig. 39. An object model asserting that class X is a subtype of class Y.

This mapping establishes a relationship between objects of X and objects of the supertype Y. Intuitively, when an object x of class X is treated as if it were an object of class Y, then there must be some object of class Y that x behaves like. The *simulates* mapping specifies this relationship. There are several constraints on the *simulates* mapping that must always hold.

Simulates must be *bistrict* [20]. Bistrictness requires that error objects always simulate error objects, and that no other object simulates an error object. Formally,

$$simulates(err_X) \overset{\text{def}}{=} err_Y, \tag{17}$$

$$(simulates(x) = err_Y) \Rightarrow (x = err_X). \tag{18}$$

Another fundamental property of *simulates* is that every object of class X simulates *some* object of class Y. That is, *simulates* must be a total function. However, this property is ensured by the semantics of LSL specifications; LSL semantics is based on total algebras, so all specified functions are assumed to be total.

The most important constraint on *simulates* is the so-called *substitution property*. The substitution property asserts that for any operation defined on Y, X-objects can be substituted where Y-objects are expected and the behavior of the operation will be unchanged (in general). Formally, it is necessary to merge the signature of the supertype, as well as the constraints on the supertype Y, into the signature and constraints of the subtype X. The *simulates* mapping provides the tool for ensuring that the inherited operations and constraints of the subtype are consistent with those of the supertype.

The simplest inherited component is the state valuation function, '\$,' defined for each class. Suppose that an X-object, x, is substituted where a Y-object is expected, and the state of x is queried. Since x is being observed as if it were a Y-object, the state valuation function must respond with an object from *Y–STATES*. Hence, if the state of an object x is $\$(x)$, then it must be determined what the state of the corresponding Y-object, *simulates*(x), should be. In order to specify this relationship, we must map the *X–STATES* to *Y–STATES*, via the *simulates* mapping:

$$simulates : X\text{–}STATES \rightarrow Y\text{–}STATES.$$

The actual definition of this mapping must satisfy two constraints:

$$simulates(undef_X) \overset{\text{def}}{=} undef_Y, \tag{19}$$

$$(\forall x : X\ .\ \$(simulates(x)) = simulates(\$(x))). \tag{20}$$

Informally stated, the first assertion requires that undefined states simulate undefined states (analogous to the relationship between error objects). The second assertion is a commutativity property; if one considers the supertype object that x simulates and observes its state, then this state is simulated by the state of x. These constraints represent a typical property used in algebra for characterizing compositional mappings, one that is generally depicted by a *diamond diagram* [27], as shown in Fig. 40. The shading indicates that the diagram *commutes*, which simply means that (20) holds.

Attributes of a supertype are handled in a similar manner. If class Y has an attribute a of class A, as depicted in Fig. 41, then every subtype X of Y must also have the same attribute. That is, we add the following signature to the specification for X:

$$a : X \rightarrow A. \tag{21}$$

The possible mappings that a may define are constrained by the *simulates* mapping, as follows:

$$\left(\forall x : X\ .\ a(x) \overset{\text{def}}{=} a(simulates(x))\right). \tag{22}$$

Pictorially, this latter constraint ensures that the diamond diagram in Fig. 42 commutes.

Subtyping obeys substitution; that is, the associations in which a class Y is referenced must be applicable when objects of Y are substituted for objects of a subtype X of Y. For each association involving Y, we induce the same association on every subtype X of Y, with X substituted for Y. Consider the

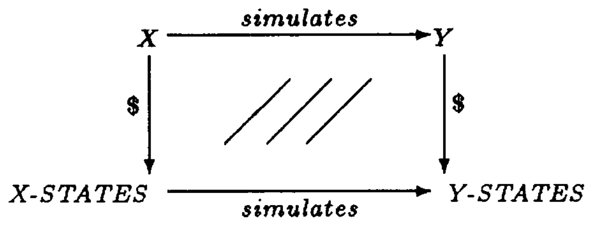


Fig. 40. A commutative diagram showing the consistency between '\$' and *simulates*.

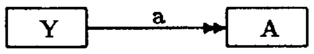


Fig. 41. An attribute, a, that must by "inherited" by all subtypes of Y.

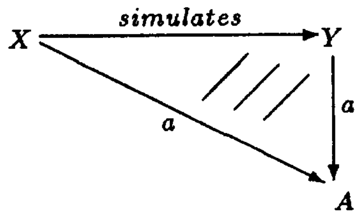


Fig. 42. The semantics of the inherited attribute a must commute.

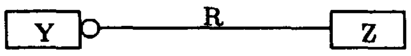


Fig. 43. An association R that is inherited by any subtypes of Y and Z.

association R shown in Fig. 43. R is an association between objects of classes Y and Z, with the following signature.

$$R : Y, Z \rightarrow BOOL. \tag{23}$$

The Y-endpoint of this association asserts that for every Z-object, there is at most one Y-object associated via R. Intuitively, one would expect that this multiplicity constraint would apply to *all* Y-objects, including those that are nominally of a subtype of Y. We formalize this intuition with an additional constraint on the *simulates* mapping, specifically, we require that *simulates* be injective. For each subtype X of Y,

$$\left(\forall x_1, x_2 : X \,.\, simulates(x_1) = simulates(x_2) \Rightarrow x_1 = x_2\right). \tag{24}$$

In order to allow X-objects to be used in place of Y-objects in the association R given in (23), a new predicate must be introduced to represent the inherited association R between X and Z. Formally, we introduce a new predicate R by the following signature:

$$R : X, Z \rightarrow BOOL. \tag{25}$$

The predicate R given above is distinct from the predicate whose signature is given in (23), but because of the subtype relationship between X and Y, the two predicates are tightly coupled. The semantics of the R named above in (25) is constrained by the semantics of the association R specified in (23). This constraint relationship is achieved by using the *simulates* mapping:

$$\left(\forall x : X, z : Z \,.\, R(x,z) = R(simulates(x), z)\right).$$

When viewed algebraically as an X-object, x is related (via R) to those Z-objects to which it relates when viewed as a Y-object; this constraint ensures consistency. Also, note that this latter constraint imposes all of the constraints regarding the association R given in (23) on the inherited association given in (25).

For the purposes of deriving specifications, it is assumed that there exist classes $V_1, \ldots, V_k$ that have no supertypes. These classes are referred to as the *base types*. Beginning with the base types, the specifications for each class in an object model are derived. Next the direct subtypes of the base types are considered, and so on, in a breadth-first manner. Since the subtype relationship is a transitive one, this approach to derivation will compute specifications that represent the transitive closure of the subtype relationship. Since the order in which class specifications are derived is important when subtyping is used, we present a simple algorithm for class specification derivation in Fig. 44, that follows the above discussion. There are two loops in the algorithm, the outer loop executing until the queue *clssq* is

```
Algorithm: Derive Class Specifications
    Input: Let $V = \{V_1, \ldots, V_k\}$ be the base types of object
    model $\mathcal{O}$. Let clssq be a queue of class names which initially
    contains the classes of $V$ in any order.
    Output: Formal specifications of classes
    Procedure:
      let clssq ← elements of V in any order.
      while clssq is not empty do
          let c ← next element on clssq, element is dequeued.
          let stypes ← $\{d_1, \ldots, d_k\}$ where each $d_i$ is a     .
          for each $d \in$ stypes
              if d is not in clssq $\wedge$ specification(d) has not
              yet been computed,
              then enqueue d on clssq.
          endfor
          compute specification(c)
      endwhile
```

Fig. 44. A simple algorithm for deriving the class specification.

empty, and the inner loop adds elements to the queue. The algorithm will terminate under two conditions: every class must have a finite number of subtypes, and the object model contains no cycles consisting solely of subtype edges. Both of these conditions are valid assumptions.

As an example derivation, consider the object model given in Fig. 46, which is different from previous models of the same objects. In this figure, objects of the class Message are depicted to have an attribute *txno*. Furthermore, Message objects are associated with User objects via the *sends* association. Since RequestMsg is depicted as a subtype of Message, RequestMsg objects have the same properties. In order to derive the specification for this diagram, we begin with the base type classes. From this figure, it appears that classes User, Message, TransactionNumber, and Mailbox are the base types, so we will treat them as such. The specifications for Mailbox and User were given earlier in Figs. 21 and 25, respectively. The specification for TransactionNumber would be similar to that of User and is assumed to exist for our current purposes. The specification for Message is given in Fig. 45.

The class RequestMsg is specified as a subtype of the class Message. This assertion requires that the attribute *txno* be inherited by RequestMsg, and that an injective simulation relation be introduced. The complete specification for RequestMsg is given in Fig. 47.

Finally, the specification for the association *sends* is derived according to the usual rules for binary association. Since *sends* associates the class User with the class Message, *sends* must also associate User with any subtypes of Message, specifically the class RequestMsg. Hence, two signatures for *sends* are introduced, one for Message, and the other for RequestMsg. The resulting specification for *sends* is given in Fig. 48.

E. Semantics of an Object Model Diagram

Having demonstrated the method by which the semantics of the basic object model constructs are defined, it remains only to describe the meaning of an object model as a whole. It is straightforward to address this last semantic issue.

```
CLASS-Message : trait
    includes
        TransactionNumber
    introduces
        undef_Msg   :  →Message-STATES
        err_Msg     :  →Message
        $           : Message →Message-STATES
        txno        : Message →TransactionNumber
    asserts
        Message-STATES generated by undef_Msg
        $(err_Msg) == undef_Msg
        txno(err_Msg) == err_TN
```

Fig. 45. Trait specification for the class Message.

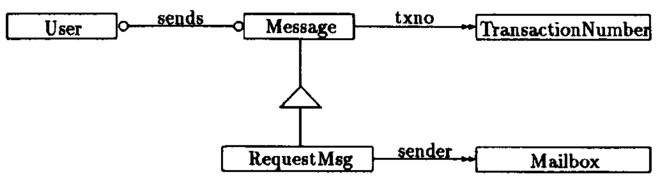


Fig. 46. RequestMsg inherits attribute *txno* and association *sends* is extended.

```
CLASS-RequestMsg :  trait
    includes
      CLASS-Message,
      CLASS-Mailbox,
      CLASS-TransactionNumber
    introduces
      undef_RQ  :     →RequestMsg-STATES
      err_RQ    :     →RequestMsg
      $         :   RequestMsg →RequestMsg-STATES
      txno      :   RequestMsg →TransactionNumber
      simulates:    RequestMsg →Message
    asserts
      RequestMsg-STATES generated by undef_RQ
      ∀r,r_1 :RequestMsg
         $(err_RQ) == undef_RQ
         %
         % bistrictness
         %
         simulates(err_RQ) == err_Msg
         (simulates(r) = err_Msg) ⇒ (r = err_Msg)
         %
         % constraining an inherited operation
         %
         txno(r) == txno(simulates(r))
         txno(err_RQ) == err_TN
         %
         % injectiveness
         %
         (simulates(r) = simulates(r_1)) ⇒ (r = r_1)
```

Fig. 47. Trait specification for class RequestMsg, with subtyping.

The semantics of an object model O is the set of instance diagrams that are consistent with O. Applying the approach to formalization described that has been described in this paper, we reduce the object model O to an algebraic specification and its instance diagrams to algebras. Hence, the semantics of O is the set of algebras consistent with the algebraic specification for O. The algebraic specification of an object model O is a single trait, whose name is *OBJECT-MODEL-O*, and which is

```
ASSOCIATION-sends :  trait
    includes
      CLASS-User,
      CLASS-Message,
      CLASS-RequestMsg
    introduces
      sends  :  User, Message → BOOL
      sends  :  User, RequestMsg → BOOL
    asserts
      ∀u, u_2 : User,  m, m_2 : Message, r : RequestMsg
         sends(err_User, m) ∧ u ≠ err_User ⇒ ¬sends(u, m)
         sends(u, err_Msg) ∧ m ≠ err_Msg ⇒ ¬sends(u, m)
         % injective
         (sends(u, m) ∧ sends(u_2, m)) ∧ m ≠ err_Msg ⇒
             u = u_2
         % functional
         (sends(u, m) ∧ sends(u, m_2)) ∧ u ≠ err_User ⇒
             m = m_2
         % subtyped inducements
         sends(u, r) == sends(u, simulates(r))
```

Fig. 48. Trait specification for *sends* association, with subtyping.

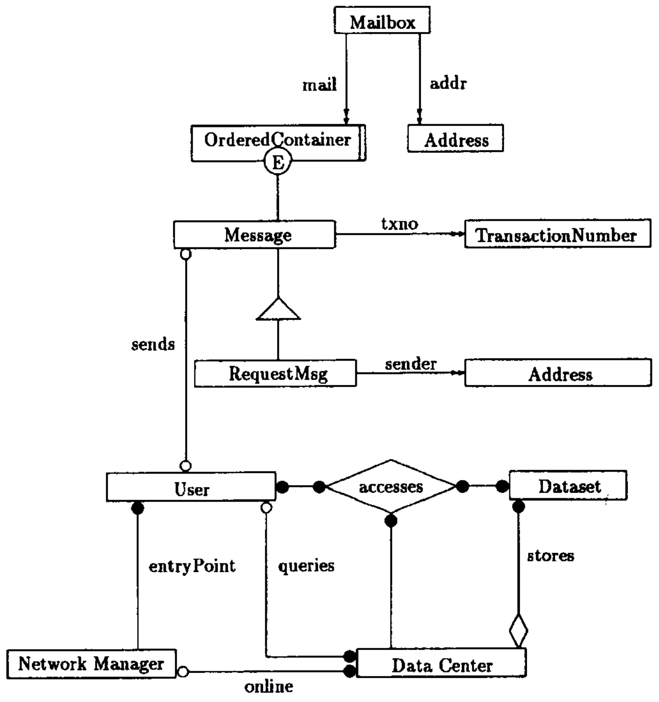


Fig. 49. The composition of several simpler object models, O_2.

simply the inclusion of all class and association traits derived from O. We illustrate this idea with a simple example.

Many object models have been introduced in this paper, but all of them describe elements of the same problem. Fig. 49 gives object model O_2 that is constructed by merging three of these object models, specifically Figs. 1, 20, and 46. The specification of this object model is given in Fig. 50. It is understood that the traits being included in this figure are constructed according to the rules described in this paper. The reader will notice that the subtype relationship and the attribute relationships are not explicitly represented in the specification given in Fig. 50, but are explicitly described in

```
OBJECT-MODEL-O2 : trait
    includes
        CLASS-User,
        CLASS-Dataset,
        CLASS-NetworkManager,
        CLASS-DataCenter,
        CLASS-Message,
        CLASS-Address,
        CLASS-RequestMsg,
        CLASS-Mailbox,
        CLASS-TransactionNumber,
        ASSOCIATION-entryPoint,
        ASSOCIATION-queries,
        ASSOCIATION-online,
        ASSOCIATION-accesses,
        ASSOCIATION-sends,
        ASSOCIATION-hasPart_DataCenter_Dataset
```

Fig. 50. Trait specification for object model O_2.

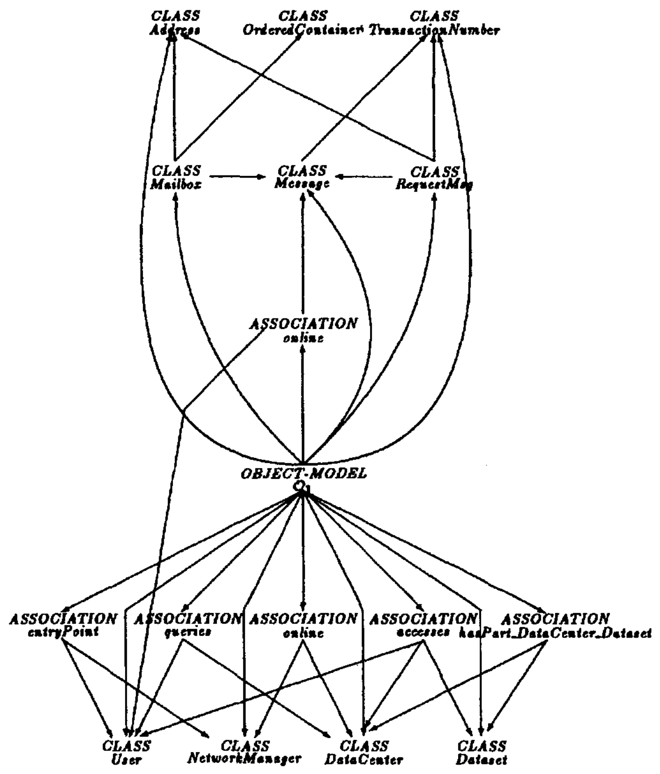


Fig. 51. A dependency graph of O_2.

the respective class specifications themselves. In addition, the class OrderedContainer has no specification; its specification is included (with parameter bound) by the Mailbox class.

Fig. 50 appears to be simple, but the reader should note that there is a significant amount of interrelated information being described in this picture. As described throughout this paper, each class and each association results in a trait specification being derived. With the introduction of attributes and subtyping, there is a dependency relationship between classes (one class specification must include another). Each edge in the object model can be converted into a directed edge representing the direction of dependency between the classes. For object model O_2, the dependencies between classes is depicted in Fig. 51, where the class at the tail of the arrow depends on the classes at the heads of arrows.

IV. Related Work

Hartrum and Bailor give a high-level Z [28] formalization of the three OMT notations, including a subset of the OMT object model notation [29]. Their formalization of object models is designed to *guide* the development of formal specifications from object models, where it is intended for the specifier to modify the specification directly to include details about analysis and design information. Our approach, in contrast, assumes that much of the analysis process will proceed at the diagram level, and our formalization techniques, amenable to automation, can be used to obtain a formal specification of the explicit and implicit information in the diagrams. Hartrum and Bailor do not formalize instance diagrams, where, in contrast, we rely heavily on the formalization of instance diagrams to formalize the OMT semantics that relate objects models to instance diagrams.

The SAZ Project [30] combines the benefits of the Structured Systems Analysis and Design Method (SSADM) with that offered by the Z specification language. SSADM, like many other structured analysis approaches involves a process comprising several steps for capturing system requirements and design. The SSADM process makes use of diagrams (data flow and entity-relationship diagrams) and extensive documentation. However, due to the different levels of detail and abstraction described by the diagrams and documentation through the numerous steps, it can be challenging to eliminate ambiguity and consistency errors between diagrams and documents within a given stage and between stages. SAZ uses the Z specifications as a means to capture the information from the diagrams and the documentation in a single integrated framework. Z specifications are used to model the functional requirements of the data in terms of state with respect to entity-relationship diagrams, thus allowing specification of subtyping. Z predicates can also be used to specify additional constraints not explicitly captured by the SSADM approach, such as conditions on data or operations. Therefore, Z adds formality to a previously informal approach offered by SSADM, thus enabling consistency checks and ambiguity detection and removal from the diagrams and the documentation. There are no explicit rules for constructing the Z specifications, since the Z schemas are largely based on domain specific information.

For the Yourdon modeling approach, Semmens and Allen [31] have developed a Z syntax for the three diagrams (DFDs, ERs, and statecharts) and the data dictionary. They developed Z functions that map Z representations of diagrams to type definitions, state and operation schemas. Using their Z-based syntax, they are able to automatically generate system state and signatures of operation schemas from the diagrams and the data dictionary. They assume that the user supplies the pre- and postconditions for the the operation schemas.

In an early introduction of the OMT notations [32], Rumbaugh developed a semiformal description of the object model notation to facilitate its use and understanding. A proprietary programming language was used to describe the notation, where the semantics were not given. Rumbaugh's treatment of the notation focused largely on design- and implementation-related concepts, while the work described in this paper focuses on the use of object models to represent information at the software requirements level, which can be processed rigorously.

Hayes and Coleman [33] have used a VDM-like [34] notation to perform a preliminary formalization of a subset of the OMT notations. Multiplicity constraints are described at a high-level, but their more general mathematical properties are not explored. Their formalization of associations is biased towards a particular implementation: pointers. The formalizations developed by Coleman and Hayes focus towards a more design-specific subset of the object model notation, which allows attributes and operations to be encapsulated in a class.

V. Conclusions and Future Investigations

Our formal semantics for the OMT object model notation enables object model diagrams to be treated as formal specifications, and thereby overcomes many of the analytical limitations inherent in this otherwise informal method. Our formalization method describes a technique for deriving modular algebraic specifications directly from object model diagrams. Of particular importance is the focus on instance diagrams in providing a graphical depiction of the semantics of an object model. These results bring to bear the organizational advantages of OMT on the construction of formal specifications.

A few extensions to the OMT object model notation have been introduced in this paper. Graph-based notations for attributes and object states were introduced to avoid the implementation bias of premature design decisions. A notation for external classes was introduced to allow for the incorporation of algebraic specifications that have no graphical specification; this capability is especially useful for using complex theories from the Larch Shared Language trait handbook.

Some notations that are standard features of the OMT object model notation were given precise definitions where, previously, no such definition existed. In particular, ternary and higher-arity associations were given a semantics that is a generalization of the semantics of binary associations. The subclass notation of OMT was also given a precise semantics in this paper, a semantics that we have referred to as subtyping.

Rumbaugh indicates that the other modeling notations of OMT are integrated in the implementation phase of the software development process. Our formalization of object models allows these notations to be integrated in the specification phase since all would have well defined mathematical theories. In related investigations, we have also provided a complementary semantics for the dynamic model of OMT [35]. A prototype diagramming tool, VISUALSPECS [36], has been developed that provides automated derivation of formal specifications from object model diagrams using the rules described in this paper. This tool is being developed as part of a more comprehensive specification environment [25] that provides graphics-based browsing, syntax checking, and theorem proving support for LSL specifications using tools developed as part of the Larch project.

Future work will investigate the development of transformations of the requirements models into models for use in design. The transformations will be applied to the diagrams and reflected in the corresponding specifications. VISUALSPECS will be updated to support the new formalization rules, including rules to support the integration of the three types of diagrams used in OMT.

Appendix I
Referenced Larch Traits

The Larch Project has produced a variety of tools to support the Larch Shared Language (LSL) and Larch interface languages. In support of LSL, a syntax/sort checking tool, a TEXifier for LSL traits (LSL2TEX), and an interactive theorem prover (LP) were developed. Distributed with these tools is a significant library of traits called the LSL Handbook, one of which was used in this paper. This trait, *Set*, and one of its main support traits, *SetBasics*, is reproduced in this appendix with the help of the LSL2TEX translator.

```
SetBasics(E, C) : trait
    introduces
        {} :→ C
        insert : E, C → C
        __ ∈ __, __ ∉ __ : E, C → Bool
    asserts
        C generated by {}, insert
        C partitioned by ∈
        ∀ s : C, e, e1, e2 : E
            e ∉ s == ¬(e ∈ s)
            e ∉ {}
            e1 ∈ insert(e2, s) == e1 = e2 ∨ e1 ∈ s
    implies
        UnorderedContainer,
        MemberOp
        ∀ e, e1, e2 : E, s : C
            insert(e, s) ≠ {}
            insert(e, insert(e, s)) == insert(e, s)
        converts ∈, ∉


Set(E, C) : trait
    includes
        SetBasics,
        Natural(N),
        DerivedOrders(C, ⊆ for ≤, ⊇ for ≥, ⊂ for <, ⊃ for >)
    introduces
        delete : E, C → C
        {__} : E → C
        __ ∪ __, __ ∩ __, __ − __ : C, C → C
        size : C → N
    asserts
        ∀ e, e1, e2 : E, s, s1, s2 : C
            {e} == insert(e, {})
            e1 ∈ delete(e2, s) == e1 ≠ e2 ∧ e1 ∈ s
            e ∈ (s1 ∪ s2) == e ∈ s1 ∨ e ∈ s2
            e ∈ (s1 ∩ s2) == e ∈ s1 ∧ e ∈ s2
            e ∈ (s1 − s2) == e ∈ s1 ∧ e ∉ s2
            size({}) == 0
            size(insert(e, s)) == if e ∉ s then size(s) + 1 else
size(s)
            s1 ⊆ s2 == s1 − s2 = {}
    implies
        AbelianMonoid(∪ for ∘, {} for unit, C for T),
        AC(∩, C),
        JoinOp(∪),
        MemberOp,
        PartialOrder(C, ⊆ for ≤, ⊇ for ≥, ⊂ for <, ⊃ for >),
        UnorderedContainer
        C generated by {}, {__}, ∪
        ∀ e : E, s, s1, s2 : C
            insert(e, s) ≠ {}
            insert(e, insert(e, s)) == insert(e, s)
            s1 ⊆ s2 == s1 − s2 = {}
        converts ∈, ∉, {__}, delete, size, ∪, ∩, −, ⊆, ⊇, ⊂, ⊃
```

ACKNOWLEDGMENTS

The authors gratefully acknowledge the detailed comments made by the anonymous reviewers, which have helped greatly to improve the presentation of the paper. Enoch Y. Wang has played an invaluable role in this research, both in terms of offering constructive criticism to the formalization and in bringing the formalization to reality through his implementation of VISUALSPECS, a tool that supports the graphical construction of the object model and the automated generation of Larch specifications. Finally, the authors thank the members of the Software Engineering Research Group at Michigan State University for feedback on this project and for effort in the ENFORMS project that made use of the OMT modeling and the formalization.

For further information about this project and related SERG projects, see the web site:

`http://www.cps.msu.edu/~ chengb/serg.html.`

This work is supported in part by National Science Foundation grants CCR-9209873 and CCR-9407318.

REFERENCES

[1] R.R. Lutz, “Targeting safety-related errors during software requirements analysis,” *SIGSOFT '93 Symp. on the Foundations of Software Engineering*, 1993.

[2] R.R. Lutz, “Analyzing software requirements errors in safety-critical embedded systems,” *Proc. of IEEE Int'l Symp. on Requirements Engineering*, 1993.

[3] J.C. Kelly, J.S. Sherif, and J. Hops, “An analysis of defect densities found during software inspections,” *J. of Systems Software*, vol. 17, pp. 111–117, 1992.

[4] V. Basili and B. Perricone, “Software errors and complexity: An empirical investigation,” *Comm. of the ACM*, vol. 21, pp. 42–52, Jan. 1984.

[5] B. Boehm, “Software engineering economics,” *IEEE Transactions on Software Engineering*, vol. 10, pp. 4–21, Jan. 1984.

[6] J.M. Wing, “A specifier's introduction to formal methods,” *IEEE Computer*, vol. 23, pp. 8–24, Sept. 1990.

[7] B.H.C. Cheng, “Applying formal methods in automated software development,” *J. of Computer and Software Engineering*, vol. 2, no. 2, pp. 137–164, 1994.

[8] P. Coad and E. Yourdon, *Object Oriented Analysis*. Prentice-Hall, Inc., 2nd ed., 1991.

[9] R. Wirfs-Brock, B. Wilkerson, and L. Wiener, *Designing Object-Oriented Software*. Prentice Hall, 1990.

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

[11] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes, *Object-Oriented Development: The Fusion Method*. Object-Oriented Series, Prentice Hall, 1993.

[12] R.S. Pressman, *Software Engineering: A Practitioner's Approach*. McGraw Hill, 3rd ed., 1992.

[13] P. Chen, “The entity-relationship model: Toward a unifying view of data,” *ACM Transactions on Database Systems*, vol. 1, pp. 9–36, Mar. 1977.

[14] D. Harel, A. Pneuli, J.P. Schmidt, and R. Sherman, “On the formal semantics of statecharts,” *Proc. Second IEEE Symp. on Logic in Computer Science*, pp. 54–64, 1987.

[15] J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley Publishing Co., Inc., 1979.

[16] M. Wirsing, “Algebraic specification,” *Handbook of Theoretical Computer Science* (J. van Leeuwen, ed.), vol. 2, ch. 13, Amsterdam: Elsevier Science Publishers and MIT Press, 1990.

[17] R.H. Bourdeau, B.H. Cheng, and B. Pijanowski, “A regional information system for environmental data analysis,” *to appear in Photogrammetric Engineering & Remote Sensing*.

[18] B.H.C. Cheng, R.H. Bourdeau, and G.C. Gannod, “The object-oriented development of a distributed multimedia environmental information system,” *Proc. Sixth Int'l Conf. on Software Engineering and Knowledge Engineering*, pp. 70–77, June 1994.

[19] B. Liskov and J. Wing, “Family values: A semantic notion of subtyping,” Tech. Rep. 562, MIT Laboratory for Computer Science, 1992.

[20] G.T. Leavens and W.E. Weihl, “Subtyping, modular specification and modular verification for applicative object-oriented programs,” Tech. Rep. #92-28, Department of Computer Science, Iowa State University, Sept. 1992.

[21] J.A. Bergstra, J. Heering, and P. Klint, “The algebraic specification formalism ASF,” *Algebraic Specification* (J. A. Bergstra, J. Heering, and P. Klint, eds.), ACM Press, Addison-Wesley Publishing Company, 1989.

[22] J.V. Guttag and J. J. Horning, *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[23] J. Horning and J. Guttag, “Larch shared language checker.” Private communication.

[24] S. Garland and J. Guttag, “A guide to lp, the larch prover,” Technical Report TR 82, DEC SRC, Dec. 1991.

[25] M.R. Laux, R.H. Bourdeau, and B.H.C. Cheng, “An integrated development environment for formal specifications,” *Proc. Int'l Conf. on Software Engineering and Knowledge Engineering* (San Francisco, Calif.), pp. 681–688, July 1993.

[26] C.-L. Chang and R.C.-T. Lee, *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, 1973.

[27] L.C. Grove, *Algebra*. Academic Press, 1983.

[28] J. Spivey, *The Z Notation, A Reference Manual*. Prentice-Hall International (UK) Ltd., 1989.

[29] T.C. Hartrum and P.D. Bailor, “Teaching formal extensions of informal-based object-oriented analysis methodologies,” *Proc. Computer Science Education*, pp. 389–409, 1994.

[30] F. Polack, M. Whiston, and K. Mander, “The SAZ project: Integrating SSADM and Z,” *FME '93: Industrial-Strength Formal Methods* (J. Woodcock and P. Larsen, eds.), Odense, Denmark, pp. 541–557, Springer-Verlag, Apr. 1993.

[31] L. Semmens and P. Allen, “Using Yourdon and Z: An approach to formal specification,” *Proc. Fifth Annual Z User Group Meeting*, Oxford: Springer-Verlag, Dec. 1991.

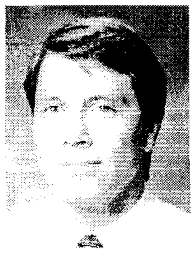
[32] J. Rumbaugh, “Relations as semantic constructs in an object-oriented language,” *Proc. of OOPSLA '87*, pp. 466–481, 1987.

[33] F. Hayes and D. Coleman, “Coherent models for object-oriented analysis,” *Proc. OOPSLA '91*, pp. 171–183, 1991.

[34] C.B. Jones, *Systematic Software Development Using VDM*. Prentice-Hall International (UK) Ltd., 1989.

[35] R.H. Bourdeau, E.Y. Wang, and B.H.C. Cheng, "An integrated approach to developing diagrams as formal specifications," Technical Report MSU-CPS-94-42, Michigan State University, Sept. 1994. (submitted for publication).

[36] B.H.C. Cheng, E.Y. Wang, and R.H. Bourdeau, "A graphical environment for formally developing object-oriented software," *Proc. IEEE Sixth Int'l Conf. on Tools with Artificial Intelligence*, Nov. 1994.



Robert H. Bourdeau manages the Applications, Tools, and Products Section of the Consortium for International Earth Science Information Network's Information Technology Division. Mr. Bourdeau has a BS degree in both computer science and mathematics from the University of Michigan, Flint. He received his MS degree in computer science with a minor concentration in mathematics from Michigan State University and is currently completing his doctorate in computer science. Mr. Bourdeau researches mathematical methods appropriate for the development of reliable, object-oriented, information systems and has published several papers on the subject.

Betty H.C. Cheng (S'87, M'90) is an assistant professor in the Department of Computer Science at Michigan State University. Dr. Cheng received her BS from Northwestern University and her MS and PhD degrees from the University of Illinois at Urbana-Champaign in 1987 and 1990, respectively. She conducts research in the areas of formal methods applied to software engineering and applications to parallel and distributed computing, software development environments, object-oriented development, and logic programming. Dr. Cheng is a member of the IEEE Computer Society and ACM.