EXECUTABLE AXIOMATIC SPECIFICATION USING FUNCTIONAL LANGUAGE - CASE STUDY: BASE ONTOLOGY FOR A SPATIO-TEMPORAL DATABASE

Andre U. Frank and Damir Medak
Deptartment of Geoinformation
Technical University Vienna
{frank, dmedak}@geoinfo.tuwien.ac.at

Abstract

Formal specifications are difficult to read.*Executable* specifications allow to see the behavior of the specified objects and help the domain specialist to detect errors quickly. We present here a method which allows to write axiomatic specifications which can be executed and discuss the limitations in expressive power imposed by the restriction to constructive axioms and how it can be circumvented.

The method results from practical efforts to formalize the meaning of object types for Geographic Information Systems. If such data are shared betweenorganisations, differences in the semantics become apparent and formal methods for their definition become necessary.

Most formal methods are based on first order languages. Software engineering often uses algebraic methods, but tools practically used for data exchange standard definitions are restricted to signatures and do not capture the behavior of the operations. We present here an algebraic approach using a functional programming language which includes specification of behavior and results in executable code.

The case used here to demonstrate the problem is the base for an ontology of spatio-temporal databases. The world consists of objects, which have identity. Identifiers, objects and world classes are defined as algebras with axioms. Executable models for these are given. This environment is necessary to describe objects with operation, when the essential part of the definition is the change an object can undergo. It would be difficult to write this in a first-order language.

The focus is on the capabilities of the executable functional programming language Haskell to formalize algebraic specifications; the issue is, how much of an algebraic specification can be expressed formally in an executable language and how much must be relegated to the implementation models (expressed in the same language). The example shows that a very large part - and most of the important behavior - can be captured in axioms for abstract classes.

1 Needs for Formal Specifications

Formal methods are difficult to use for the domain specialist. We propose here specifications which can be*executed*. The domain specialist can observe if the specified behavior corresponds to the intended behavior. The goal to make specifications executable restricts the expressivity of the language to constructive statements. This paper shows, using a non-trivial example, how this can be achieved with compromises which are acceptable in practical situations.

Formal models are necessary to specify the meaning of data; this is particularly important, when data are exchanged. Our area of application are Geographic

Information Systems, which are complex, large and widely used systems to manage data about the world. They are used to maintain information about real estate property rights (cadastre) (Dale and McLaughlin 1988), they are also used in town, county and national administration to manage resources and plan land use, etc.(Maguire, Rhind, and Goodchild 1991). Data are often shared and standards for the exchange of spatial data have been developed (Guptill 1991), however to capture the semantics of the object classification remains an open problem (Frank and Kuhn 1995; Kuhn 1995). It cannot be solved with natural language descriptions (Kuhn 1994; Kuhn submitted), especially not in Europe, where national languages aggravate the problem (Mark 1993). Formal methods are required.

The formal methods used to define object types are mostly based on first order languages. These focus on properties of objects and the resulting definitions have more problems with the prototype effect of human cognition (Lakoff 1987; Rosch 1973). Algebraic formalizations stress the operations an object can be involved in and the effects of such operations on the object, other objects and the relations between the objects. This seems to be more in accordance with principles of human cognition, as they can also deal with the hierarchical structures, human use (Frank 1996a; Frank 1996c; Langacker 1987). Algebraic methods are widely used in software engineering (Liskov and Guttag 1986; Woodcock and Loomes 1989). For standardization work, methods given rely on the signatures of the operations, but do not include descriptions of the effects of the operations (behavior) (ISO 1992; OGC 1994; Schenck and Wilson 1994)

In this paper a formalization method based on universal algebra (Birkhoff 1945; Birkhoff and Lipson 1970) is used for specification. This has become standard practice in software engineering and is often called 'abstract data types (Ehrich, Gogolla, and Lipeck 1989). We follow the tradition of Larch and combine small algebras (Guttag, Horning, and Wing 1985). Algebraic specifications are sometimes difficult to comprehend; a tool to check syntax and semantics is most welcome. We found that a functional programming language based on classes provides most of the properties algebraic specifications require and allows the formalization of theories in a checked syntax and their implementation in clean models which are executable.

This contribution focuses on how much of the axioms can be expressed in a constructive, executable form. It is found that only a few trivial axioms must be assumed for standardized operations linking to the models. The typical one is the axiom expressing the behavior of a pair of *get from x* and *put a in x* semantics (i.e. *get (put a x) = a*).

The example used here is the foundation for a specification of behavior of complex objects; it is taken from a proposal for a base ontology for a spatio-temporal database (TMR Project CHOROCHRONOS). In this context, definition of object properties must include the change effect by the operations on objects; simple static relations are not sufficient. A variant of this base ontology has been used to formalize the most important aspects of an administrative system, namely a land registry (Frank 1996b).

The paper starts in the next section with a discussion of the methods to write executable, algebraic specifications and gives some background about the use of functional programming languages. The following section gives the background for the case used. The next section gives formal specifications for identifiers, objects and the world in which objects live. Then section 5 comments on the methods used to include axioms in the theories.

2 Method to Write Executable Specifications

Prolog was used occasionally to write specifications. The restriction to use logic and the difficulties to use the non-logical parts of the actual Prolog language limited these efforts. Specification methods have stressed the algebraic approach(Liskov and Guttag 1986) and tools were developed (Guttag, Horning, and Wing 1985) but not widely used. We propose here to use the language Gofer (Jones 1991) (similar to Haskell (Hudak et al. 1992; Peterson et al. 1996)). It is a functional programming language with classes. It provides an advanced framework for object-oriented algebraic specification. The language is strict without side effects and code is therefore referentially transparent. Gofer has a novel class system with multiple parameters; the type inference mechanism of ML(Milner 1978) has been suitably extended(Jones 1994). It allows inheritance in the framework of 'parametric polymorphism'(Cardelli and Wegner 1985), which is easier to comprehend than other inheritance methods used in commercial programming languages.

A *class* describes an algebraic theory for a *sort* with a set of *operations* with name and signature, and axioms, which describe the behavior. A *data type* description gives the structure of representations and an *instance* description links operations from classes with the representations; they explain how the operations are carried out, using the representation as a model. It is possible, for a class to have several models **(**instantiations) and a data representation can have operations from various classes defined for them. This provides the same functionality which is described as multiple inheritance in object-oriented programming(Lochovsky 1986; Meyer 1988; Rumbaugh et al. 1991), but in the mathematically rigorous context of category theory(Asperti and Longo 1991; Barr and Wells 1990) and denotational semantics(Peyton Jones 1987; Stoy 1977).

Algebras for specification are - in our practice - very small, similar to the traits in Larch (Guttag, Horning, and Wing 1985). They are combined, as one algebra can use other algebras previously defined. The methods proposed in Larch for the combination of traits are extensive and complex. The design of a comprehensive set of methods, which can be used practically to combine individually specified traits remains an open problem. In Gofer, the sorts can be parametrized and for parameters (or combination of parameters) other algebras can be asserted. This is called *context*. It implies that the operations and axioms of the named class are available in this theory. Classes can be further combined by instantiation for the same model and operations from both algebras can be applied. There are separate means to share implementations when coding. The novel element, which makes this possible, is the clean separation between a theory (*class*) and a model (*data* and *instance*) in the code.

Gofer and Haskell are strictly typed languages, which assign a type to every object in the language (even functions have an expressible type) and check the consistency of the types in any expression automatically, using type inference. This saves us from manually checking argument types etc. and discharges automatically all axioms based on type.

Gofer (and Haskell) are functional programming languages and the code is executable. This restricts the expressive power for the axioms to constructive axioms. Example: an axiom for $\sqrt{x}$ cannot be *exist y, such that y * y = x,*

but must point to some executable strategy to calculate the square root (e.g., Newton's rule), which usually requires access to the representation and thus can only be done in a model.

When describing the model, axioms and contexts of the same format as in the classes can be used, but here the internal representation is available and thus every computable function can be expressed. In practice, much of the specification is put into the model, because it requires a representation to make it expressible. This is unfortunate, as the axioms are then less prominent, not automatically guaranteed by all models, etc. It has been asked if it is justified to call this an algebraic specification or if it should be rather called a model based (executable) specification method.

In this paper, the main emphasis of the formalization is to achieve maximum strength of axioms in the class description. In section 4 we show how the axioms can be expressed in the algebra. As a desirable effect, the implementation of the models becomes simpler and restricted to very standardized operations. Some of these are automatically produced with the newest version Haskell(Peterson et al. 1996).

Syntax

Class and data types start with capitals, variables with lower case letters. Classes list the name of the operations followed by the signature: a list of types of arguments and at the end the type of the result. Types for classes are parametrized, and multiple parameters are used to describe abstractions of types like *List of Integer* or *Tree of Float* as *f a*. Operations are defined with equations which are restricted to expressions, which can be evaluated, and pattern matching on the left-hand side.

Models consist of data types which introduce a representation, and instances which define how the operations, which were not sufficiently defined in the class, are to be carried out. The type *list* is predefined, and marked with [], the colon (:) is the concatenation *operations.*

The major difference to standard algebra is the use of parenthesis: they are only used to group terms and are not used to indicate function application; a function *f (x)* is written just *f x.*

3 The Case Studied: A World of Identifiable Objects

The world for a spatio-temporal database consists of objects which can be changed but retain their identity. This is not so much a statement about the physicalorganisation of the world (ontology), but about organisation of our perception and cognition of the world (epistemology). Typical objects are trees, apples, tables, but also animated objects, like cows, persons etc.

Internally, object identifiers are used to test objects for identity. These could be completely hidden from the user, but we provide an operation for the user to inquire about the ID of the last object stored and to access objects based on its ID (to simplify coding of complex situations).

The classes given here provide the common behavior of all entities in the model of the world. Specific sub-classes will haveparticular additional behavior, e.g., for liquids, objects manufactured from parts (Simons and Dement 1996), but eventually also objects, which depend on other objects for their existence, like shadows, holes, parcels etc.

4 Case Study: Axiomatic Specifications

To describe a world with objects, three types are needed: identifiers, objects (with the specifics of the object parametrized and left open for later determination), and the description of the world, as a collection of objects.

4.1 Identifiers

Identifiers are used internally and - in principle - need not be made available to the user at the surface. They are used to give to each represented object a unique identity and allow to test if two representations are representations of the same object or not.

The operations necessary: create a new ID (given the last ID assigned), compare if two ID are the same

```
class  (Eq i, Num' i) => IDs i where
       startId :: i
       startId = zero1
       newId :: i -> i
       newId = inc1
       sameId, notSameId :: i -> i -> Bool
       sameId i j = i == j
       notSameId i j = not (sameId i j)
```

4.1.1 Axioms needed:

- *notSame (i, newId (i))= true* – follows from the class *Num* included in the context, which has the Peano axioms, including *succ x /= x.*
- *sameId* is an equality operation (inherited from *Eq*)

The axiom for *same* uses the theory for equality defined previously (expressed as a context *Eq i =>*). Actually, the definition contains the operations *equal* and *not equal* and as a the only axiom the statement that *not equal* is the negation of *equal*. It cannot contain the standard axioms for equality, namely symmetry, reflexivity and transitivity (these can not be expressed constructively and can only be tested for any model, as discussed later). We nevertheless assume that this class has the required axioms and will use them in other classes, e.g., IDs.

```
class Eq a where
    (==), (/=) :: a -> a -> Bool
    x /= y      = not (x == y)
```

4.1.2 Model for IDs

As a model for IDs we use integers. The integers implemented by the processor are a proper model for a subset of integers axiomatically defined and could be used here. Other models would be possible.

The code only needs to state that an instance is to be formed, but all the operations are already fully defined in the class. This assures that all models follow the same axioms.

4.2 Things

The description of things of interest is using the same methods. Here only an example thing, the representation of a car is included. In this view of the world, the only property of cars is their color. Two cars are not identical if they have the same color, but in the representation, this cannot be decided and needs the functionality of objects (in the next subsection).

```
data Color = Blue | Green | Red | White
class Cars c where
       car :: Color -> c
       getColor :: c -> Color
       putColor :: Color -> c -> c
       paint :: Color -> c -> c
       paint color car = putColor color car
data Car = Car Color
```

```
instance Cars Car where
      car c = Car c
      putColor color (Car c) = Car color
      getColor (Car c) = c
```

4.3 Objects

Objects may represent arbitrary features of the world; the object class is a 'wrapper' around things to give them the desired properties all objects should have. Here only the most basic properties of objects are specified, namely identity . More specific object types will be inherited from this object type (e.g., liquids or manufactured objects consisting of parts). The object class is specified with a type parameter for the specific thing type, which this object represents. It has also a type parameter for the type of the identifier, which must be of the class IDs (as defined above).

Identity cannot always be decided based on the observable properties of an object; two different objects can appear to be equal, but are different. This results from the abstraction made for representation: only some properties are included in the model and two objects may have equal values for all of these, but still be two different objects. For example, two similar cars of the same color - if that is the only property considered - may appear equal, but they are two different objects (and differ in some other property, perhaps the make and certainly in the vehicle identification number). Object behavior requires therefore to wrap the data representing the object into a folder with a unique identifier.

The operations for objects are therefore

- *same*: to test for identity ,
- *put and get* the ID of an object and testing if a given object has a specific ID (used as a handy abbreviation), also get the thing from an object,
- *doThing*: apply a function f (e.g., paint) to the thing wrapped in the object.

```
class IDs i => Objects o t i where
      obj ::  t -> i -> o t i
      getId ::  o t i -> i
      getThing :: o t i -> t

      doThing :: (t -> t) -> o t i -> o t i
      doThing f o = obj (f (getThing o)) (getId o)

      same :: o t i -> o t i -> Bool
      same i j = sameId (getId i) (getId j)

      isId :: i -> o t i -> Bool
      isId i o = sameId i (getId o)
```

4.3.1 Axioms needed

- get/put semantics for

  getId (obj t i) = I

  getThing (obj t i) = t

These conditions can only be expressed for a model, for which substitution yields these axioms.

- equality relation for *same*: follows from *sameId* is from class ID, which is an equality relation and included in the context.

4.3.2 Model

The objects are represented as a data structure with the ID and the representation of the object. Only the three trivial operations to make an *Obj* and to get its ID or values for the thing part back, are required.

```
data Object t i = Obj t i
instance Objects Object t i where
      obj t i = Obj t i
```

```
        getId (Obj t i) = i
        getThing (Obj t i) = t
```

4.4 Worlds

The world collects the object and is the context for the uniqueness of the identifier. The world represents the state of the modeled part of the world at a specific moment in time. This model of the world is to be used as a logical model for a temporal database, where each changed world is represented as a new representation of the world (the implementation in the functional programming language, automatically shares all non-changed parts (Peyton Jones 1987)).

The operations of interest for the user are:

- *putThing,* which inserts a thing as an object into the world.
- *GetThing2,* which retrieves a thing from the world, given the ID;
- *do,* which applies a function to a thing stored in the world, given the ID.

There are a number of internal operations, which are used by those: In the representation the last ID used is kept and *nextId* is used to change this to the next one. The objects are stored in some data structure and the operations for to insert an object, to retrieve the object and to apply a function to a specific object depend on this data structure (this could be factored out).

It uses also a set of operations to make a new world (which should be empty) and get and put operations for the last *id* used and the data structure, which stores the objects; these must have the get/put semantics defined before.

```
  class (IDs i, Objects o i t) => Worlds w o i t where
        newWorld          :: w o i t
        getId2            :: w o t i -> i
        putId2            :: i -> w o t i -> w o t i
        getObjs           :: w o t i -> [o t i]
        putObjs           :: [o t i] -> w o t i -> w o t i
  -- operations on the storage structure
        putObj            :: o t i -> w o t i -> w o t i
        getObj            :: i -> w o t i -> o t i
        doObj       :: (o t i -> o t i) -> i -> w o t i -> w o t i
  -- operations with their axioms
        nextId      :: w o t i -> w o t i
        nextId w = putId2 (newId (getId2  w)) w
        putThing    :: t -> w o t i -> w o t i
        putThing t w = putObj (obj t i) w'
              where
                          w' = nextId w
                          i = getId2 w'
        getThing2   :: i -> w o t i -> t
        getThing2 i w = getThing (getObj i w)

        do                :: (t->t) -> i -> w o t i -> w o t i
        do f i w = doObj (doThing f) i w
```

4.4.1 Example for a data structure for the world

The instance of the world is a data structure, consisting of the last ID value used and a list of objects. The axioms for the *put* and *get* operations and the operations which depend on the particulars of the data structure are expressed here. The data structure could be separated in a different class, with abstract axioms; this is not done here to keep the case simple. The data structure would become an additional parameter and models with different data structures would become possible.

```
  data World o t i = W i [o t i]
  instance (IDs i, Objects o t i) => Worlds World o t i where
        newWorld = W startId []
        getId2 (W i os) = i
        putId2 i (W j os) = W i os
        getObjs (W i os) = os
        putObjs os (W i os2) = W i os

        putObj o (W i os) = W i (o:os)
        getObj i (W i' stuff) = get' i stuff
          where
          get' i [] = error "not found"
          get' i (s:ss) = if isId i s then s else get' i ss
    doObj f i (W j os) = W j [if (isId i x) then f x else x| x <- os]
```

4.4.2 Axioms needed

- put/get semantics for *putThing*, *getLast* etc.
- new collection of semantics for newWorld:
  for any x : contains (x, new) = false

This is inherited in the model from the list, where [] marks the empty list, which contains no element.

- unique identifiers within world. Follows in the model from the use of the *putThing* operation, which is always called in connection with *newId.*
- *getThing2* needs a get/put semantics for collections (follows from the axioms for the data structure, here not elaborated)
- the behavior of the *do* operations depends on the traversal of the collection, here with a list comprehension.
  f(getThing2 i w) = getThing2 i (do f i w)

5 Observations about the Placement of Axioms

In functional programming it is customary to place most 'function definitions' in the instances, because it is easier to write the executable expressions. Moving to an algebraic, object-oriented style, axioms should be placed in the class (theory) descriptions. This is beneficial, as the implementation of a model is smaller and simpler (only the put and get operations need to be defined) and the axioms apply automatically to all models.

To express the axioms in the theory (class) part is possible. The class must contain some basic operations, which put and retrieve data values to and from the representation. For these operations, a get/put semantics is assumed. The semantically more important axioms can then be given as equations using these functions.

The coding becomes more general and requires more type parameters for the classes. This tests the limits of the inference mechanism for type parameters (Jones 1995).

6 Conclusions

Formal specifications are necessary to define the semantics of objects such that they do not depend on subjective interpretation. Formal definitions of semantics are important in many areas, for example when exchanging data between different organisations. Standardization has used descriptions of properties, but more effective solutions are based on descriptions of the operations applicable to the objects defined.

We propose to use an algebraic formalization to capture the semantics. Operations are given and the effects of the operations are explained in terms of defined (observer) operations. Functional programming languages can be used for writing specifications. They provide tools to check syntax. They also allow the execution of these as computational models. To check formal specifications for intended behavior is difficult, but humans can easily observe if a demonstrated model shows the correct behavior. It becomes then possible to observe the behavior specified and errors in the specifications can be discovered quickly even by non-experts in formalization. This is a very substantial advantage for writing practical specification.

This paper has explored a particular functional programming language to understand how much of the theory can be expressed constructively in an executable syntax. It was found that most of the important aspects of the theories can be captured in algebraic axioms. Only minor assumptions must be relegated to the models and the basic implementation of the execution engine.

Acknowledgments

We are indebted to Mark Jones for the theory and the Gofer language, which makes this all possible. The presentations in(Jeuring and Meijer 1995) have contributed to my understanding of functional programming. Werner Kuhn has posed the important question, which triggered this paper, and has commented on an earlier version. We appreciate Roswitha Markwart's efforts to make the text clear and easy to read.

References

Asperti, A., and G. Longo. 1991. *Categories, Types and Structures - An Introduction to Category Theory for the Working Computer Scientist*. Edited by G. M. a. M. Albert, *Foundations of Computing*. Cambridge, Massachusetts: The MIT Press.

Barr, M., and C. Wells. 1990. *Category Theory for Computing Science. Englewood Cliffs, NJ*: Prentice Hall.

Birkhoff, G. 1945. Universal Algebra. In Proceedings of First Canadian Mathematical Congress.

Birkhoff, G., and J. D. Lipson. 1970. Heterogeneous Algebras. *Journal of Combinatorial Theory* 8:115-133.

Cardelli, L., and P. Wegner. 1985. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys* 17 (4):471 -522.

Dale, P. F., and J. D. McLaughlin. 1988. *Land Information Management*. Oxford: Oxford University Press.

Ehrich, H.-D., M. Gogolla, and U.W. Lipeck. 1989. *Algebraische Spezifikation abstrakter Datentypen.* Edited by H.-J. Appelrath, V. Claus, et al. *Leitfäden und Monographien der Informatik*. Stuttgart: Teubner.

Frank, A. U. 1996a. Hierarchical Spatial Reasoning. Internal Report: Dept. of Geoinformation, Technical University Vienna.

Frank, A. U. 1996b. An object-oriented, formal approach to the design of cadastral systems. In Proceedings of 7th Int. Symposium on Spatial Data Handling, SDH '96, at Delft, The Netherlands.

Frank, A. U. 1996c. Using Hierarchical Spatial Data Structures for Hierarchical Spatial Reasoning. Internal Report.: Dept. of Geoinformation, Technical University Vienna.

Frank, A. U., and W. Kuhn. 1995. Specifying Open GIS with Functional Languages. In *SSD'95 Proceedings*, edited by M. Egenhofer and J. Herring. New York: Springer-Verlag.

Guptill, S. C. 1991. Spatial data exchange and standardization. In *Geographical Information Systems: principles and applications*, edited by D. J. Maguire, M. F. Goodchild and D. W. Rhind. Essex: Longman Scientific & Technical.

Guttag, J.V., J. J. Horning, and J. M. Wing. 1985. Larch in Five Easy Pieces: Digital Equipment Corporation, Systems Research Center.

Hudak, P., et al. 1992. Report on the functional programming language Haskell, Version 1.2. *SIGPLAN Notices* 27.

ISO. 1992. The EXPRESS language reference manual, ISO TC 184.

Jeuring, J., and E. Meijer, eds. 1995. *Advanced Functional Programming.* Vol. 925, *Lecture Notes in Computer Science.* Berlin: Springer-Verlag.

Jones, M. P. 1991. An Introduction to Gofer: Report, Yale University.

Jones, M. P. 1994. *Qualified Types: Theory and Practice, Ph.D Dissertation, Programming Research Group, Oxford University*: Cambridge University Press.

Jones, M. P. 1995. Functional Programming with Overloading and Higher-Order Polymorphism. In *Advanced Functional Programming*, edited by J. Jeuring and E. Meijer. Berlin: Springer-Verlag.

Kuhn, W. 1994. Defining Semantics for Spatial Data Transfers. In Proceedings of 6th International Symposium on Spatial Data Handling, at Edinburgh, UK.

Kuhn, W. 1995. *Semantics of Geographic Information*. Edited by A. U. Frank. Vol. 7, *Geoinfo Series.* Vienna, Austria: Dept. of Geoinformation, TU Vienna.

Kuhn, W. submitted. Approaching the Issue of Information Loss in Geographic Data Transfers. *Geographical Systems.*

Lakoff, G. 1987. *Women, Fire, and Dangerous Things: What Categories Reveal About the Mind.* Chicago: University of Chicago Press.

Langacker, R. W. 1987. *Foundations of Cognitive Grammar*. Vol. 1 - Theoretical Prerequisites. Stanford, CA.: Stanford University Press.

Liskov, B., and J. Guttag. 1986. *Abstraction and Specification in Program Development*. Cambridge, Mass: MIT Press.

Lochovsky, F. (Ed.). 1986. *Object-Oriented Systems.* Database Engineering. Vol. 8, No. 4.

Maguire, D., D. Rhind, and M. Goodchild, eds. 1991. *Geographic Information Systems: Principles and Applications*. London: Longman Co.

Mark, D. M. 1993. Toward a Theoretical Framework for Geographic Entity Types. In *Spatial Information Theory: Theoretical Basis for GIS*, edited by A. U. Frank and I. Campari. Berlin: Springer-Verlag.

Meyer, B.. 1988. *Object-Oriented Software Construction.* New York, NY: Prentice Hall.

Milner, R. 1978. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences* 17:348-375.

OGC. 1994. OGIS Augments Data Transfer. *GIS World.*

Peterson, John, Kevin Hammond, et al. 1996. Report on the functional programming language Haskell, Version 1.3. *http://haskell.cs.yale.edu/haskell-report/haskell-report.html.Yale University Research Report YALEU / DCS/ RR-1106*.

Peyton Jones, S.L. 1987. *The Implementation of Functional Programming Languages*: Prentice Hall International.

Rosch, E. 1973. Natural categories. *Cognitive Psychology* 4:328-350.

Rumbaugh, J., et al. 1991. *Object-Oriented Modeling and Design.* Englewood Cliffs, NJ: Prentice Hall.

Schenck, D., and P. Wilson. 1994. *Information Modeling: The EXPRESS Way*. New York: Oxford University Press.

Simons, P. M., and C. W. Dement. 1996. Aspects of the mereology of artifacts. In *Formal Ontology*, edited by R. Poli and P. Simons. Dordrecht, The Netherlands: Kluwer Academic Publishers.

Stoy, J. 1977. *Denotational Semantics*. Cambridge, Mass: MIT Press.

Woodcock, J., and M. Loomes. 1989. *Software Engineering Mathematics, SEI Series in Software Engineering*. New York, NY: Addison-Wesley.