UC Santa Barbara
NCGIA Technical Reports

Title

Formalizing Behavior of Geographic Feature Types (95-7)

Permalink
https://escholarship.org/uc/item/52s6113c

Authors
Rugg, Robert D.
Egenhofer, Max J.
Kuhn, Werner

Publication Date
1995-06-01

eScholarship.org Powered by the California Digital Library
University of California

Formalizing Behavior of Geographic Feature Types

by

Robert D. Rugg, Max J. Egenhofer, and Werner Kuhn

National Center for Geographic Information and Analysis

Report 95-7

Copyright

© National Center for Geographic Information and Analysis

June 1995

Table of Contents

Abstract

Acknowledgments

1. Introduction

2. The Entity Type DAM in SDTS

3. Functional Algebras for Entity Type Definitions

4. A Definition of Dam using Functional Algebra

4.1 Constructing a Dam

4.2 Raising a Dam

4.3 Reservoir is Full

4.4 Lowering a Dam (discharging)

5. Conclusions

5.1 Summary

5.2 Discussion

5.3 Future Work

6. References

Abstract

This paper addresses the problem of formalizing the natural-language definitions of spatial features. While the Spatial Data Transfer Standard (SDTS) supports the structural aspects of the definition of spatial features, it falls short of providing means to convey explicitly their behavior. An approach using functional algebra is developed using the example of the SDTS standard entity types "dam," "watercourse," and "lake," together with the operations expressed in the natural-language definitions of these features. Formal algebraic specifications go beyond the SDTS approach, by providing precise mathematical representations of the behavior of geographic features and the interactions among related feature types. Functional specifications also help in refining the selection of attributes needed to characterize the behavior of a given feature type. An implication of the functional approach is to provide precise mathematical signatures of feature types as an alternative to natural-language definitions. Mathematical specifications are unambiguous across cultures and languages and provide a strict basis for assessing the interoperability of objects in feature-based GISs.

Acknowledgments

This work was partially supported by the National Science Foundation (NSF) for National Center for Geographic Information and Analysis (NCGIA) under grant number SBE-8810917. Bob Rugg's work was supported under the visiting scholars program of the National Center for Geographic Information and Analysis (NCGIA) at the University of Maine and by a sabbatical leave from the Department of Urban Studies and Planning, Virginia Commonwealth University. Max Egenhofer's research is further supported by NSF grant IRI-9309230, and grants from Intergraph Corporation, Space Imaging Inc., Environmental Systems Research Institute, and the Scientific and Environmental Affairs Division of the North Atlantic Treaty Organization.

1. Introduction

Recently, it has been suggested that even an extended *entity-relationship* model, i.e., one that provides for specified relationships between types of geographic features[1], would be insufficient for data exchange in which the meanings associated with feature types are intended to be well defined and preserved. This is because the meaning conveyed in the definition of an entity type includes dynamic concepts that involve how the entity in question will function in a given situation. Operations in this sense are poorly modeled, and only indirectly observed, in a system based on the static concepts of entity, attribute, and relationship. Frank and Kuhn (1995) note:

> Some data definition languages (e.g., EXPRESS [ISO, 1992]) allow to specify data types, but lack formal semantics. They describe static data types with attributes and relationships, omitting the specification of operations. However, a specification language based on types must have a method to associate data types with operations. Otherwise the concept of type remains vacuous.

An example is using a road (a data type) for vehicular traffic (an operation). During the winter season, in Maine, people cross frozen lakes and rivers, in cars and snowmobiles, as if they were roads. The same operation would be impossible in the summer time, and hazardous at other times of year. Is the frozen lake a "road?" If we specify the operations that are the critical properties of a road, for instance that it handles vehicular traffic, we have a more robust understanding of this type of feature than if we are content to classify things by static attributes like size, shape, and composition, or by static relationships such as the road crossing above the river on a bridge.

Frank (1994), Frank and Kuhn (1995), Kuhn and Frank (1991), Mark (1993), and others have proposed using functional algebra as a technique for modeling spatial data. Frank and Kuhn (1995) assert that functional language data type specifications for "open GIS" serve to describe differences in semantics of geographic information systems (GIS) operations. Kuhn (1994) proposed that this technique can be applied directly to the problem of defining feature types for spatial data transfers. For example, he pointed out that the SDTS definition for DAM, "a barrier constructed across a watercourse to control the flow or raise the level of water" (SDTS: 9) includes phrases that describe how a dam functions. This natural-language definition suggests operations of the entity type DAM, namely changeFlow (of water) and changeLevel (of water). Thus, he was able to replicate the SDTS definition precisely, using a functional specification language.

Kuhn also showed that by examining the operations suggested in the definitions, the German term "Damm" is closer in meaning to the SDTS entity type EMBANKMENT than it is to the SDTS DAM. Although it appears that the German "Talsperre" is a close synonym of DAM, it is entirely possible that there is no precise German equivalent of an entity type that was originally defined in English. Thus, in order for the full meaning of the definition of DAM to be preserved in a transfer, particularly an international transfer, the model used for defining entity types must include operations as well as attributes and relationships. An entity-attribute-relationship model can be used to record the knowledge

---

[1] We use the term *feature* in the SDTS sense: the combination of a real world entity type and its representation as an object in a database. When we use the term *entity type*, we mean specifically the classification of real world phenomena as presented in Part 2 of SDTS.

that the DAM has a certain HEIGHT and DISCHARGE, that the WATERCOURSE has a depth and a FORCE_OF_FLOW, and that the DAM has a relationship of CONTROL_OVER_WATER_LEVEL to the WATERCOURSE. However, even knowing all this, we still do not know, other than from the natural-language definition, that the basic purpose of a DAM, i.e., its *behavior*, is to control the flow and raise the level of water.

Spatial data transfer and data sharing within the emerging Global Information Infrastructure will require much tighter control over the semantics of the data being shared. We need methods that will enable producers of data to specify the full meaning of what they are providing and that will enable data users to understand exactly what information they are receiving. Otherwise, we will never have *information highways*, but only continue to have *bit-string highways* (Zemankova, 1995).

In the following sections, we will extend Kuhn's example by providing a more exhaustive definition of the entity type “dam,” first using the current types, definitions, attributes, and capability to model relationships found in SDTS, and then using functional algebra to specify the behavior of the dam.

2. The Entity Type DAM in SDTS

Part 2 of SDTS is based on a “Conceptual Model” of spatial features (SDTS: 2) consisting of entity types, entity instances, attributes, and attribute values. It is assumed that all natural phenomena can be described by applying these concepts.

The concept of *relationships* between entity types is not expressly included in the SDTS model, although it is possible to represent such relationships as *attributes* of one entity for which the *attribute value* is the related entity. Part 2 of SDTS also includes a large number of *included terms*, for approximately 1,200 entity types with definitions that overlap those of one of the 200 standard entity types defined in the standard. Often, an included term is a subcategory of a standard term. For example, *pond* is an included term under the standard term *lake*. Standard attributes are defined in SDTS to help distinguish the nuances of meaning between a standard term and an included term. For example, *size* is an attribute and so a pond can be encoded into SDTS as a *small lake*.

There is no logical reason why all the necessary information about real world features cannot be represented within the framework devised for SDTS. However, critics have argued, among other things, that

- relationships among entity types should be an explicit part of the SDTS model;
- natural-language definitions are too vague and unreliable to be consistently used and understood by diverse individuals, particularly if the individuals are in different organizations, live in different cultures, or, worse, speak different natural languages; and
- even if the meaning of a definition is clearly understood, the information about a particular slice of real world entities provided for a particular application may be of limited or no use in other applications.

We will return to these criticisms at the conclusion of this paper. First, we review how a standard entity is recorded in SDTS, using an extended version of Kuhn's example.

In SDTS, a DAM is a standard entity type. The water controlled by the DAM, in SDTS, flows through a WATERCOURSE. A related term is RESERVOIR, the standing body of inland water backed up behind a dam. In SDTS, RESERVOIR is an included term under the standard term LAKE, the definition for which should read "any standing body of inland water." These are defined in SDTS as follows:

DAM	A barrier constructed across a watercourse to control the flow or raise the level of water.
WATERCOURSE	A way or course through which water may or does flow.
LAKE	Any standard [sic] body of inland water.

Several SDTS standard attributes may be used to further describe the DAM, WATERCOURSE, and (or) LAKE and the relationships among them. For example, each can have a NAME and a WIDTH. The DAM has a HEIGHT and DISCHARGE and is characterized by CONTROL_OVER_WATER_LEVEL of the WATERCOURSE. It may be intended for FLOOD_CONTROL, HYDROELECTRIC_POWER, or other purposes. It is usually MANMADE and may be EXISTING or PROPOSED. The WATERCOURSE has a DIRECTION_OF_FLOW, WETTED_PERIMETER, CROSS_SECTIONAL_AREA, FORM_RATIO, and FORCE_OF_FLOW. It may be INTERMITTENT or PERENNIAL and may be RECREATIONAL. It is related to the DAM as a FEATURE_PRESENT. The reservoir LAKE has an AREA, a VOLUME, a SALINITY, and a SEASONAL_DEPTH and is MANMADE. It may be EXISTING or PROPOSED and may be RECREATIONAL, intended for WATER_SUPPLY or IRRIGATION, and subject to use RESTRICTIONS. It is related to the DAM and the WATERCOURSE each as a FEATURE_CONNECTED (Table 1). An instance of each entity type would be coded in SDTS as a tuple with each field containing an attribute value (Table 2).

Each of the entity types and attributes in SDTS has a standard definition. However, there is no standard linkage between entities and attributes. In the above examples, the attributes listed for each entity type are available to be used, but are not required. As a result, any two encoders of SDTS entity and attribute information will likely arrive at different attribute schemas for any given entity type. Even more importantly, attributes that are crucial for modeling the behavior or operation of a particular entity type may be omitted from the data transfer. Finally, even this relatively simple example shows that the relationships among various entity types are a critical part of the information to be conveyed. A dam cannot exist without a corresponding watercourse, nor a reservoir without both. Once a dam is included, we also need an attribute of the watercourse that indicates how its level is affected by the dam. The previously proposed attribute "depth" could serve this purpose.

A more important limitation of the current SDTS model is that the operations of (or affecting) entity types are not specified. If the Orono Town Dam were to be used in a hydrologic model of the Stillwater River, for example, the model and certain properties of the dam would have to be transferred outside SDTS as part of a "private agreement" between the data producer and the user. On the other hand, if the data were only being used as part of an inventory of the locations of and dimensions of existing dams, the SDTS description might be entirely sufficient.

SDTS Entity Type	*SDTS Attribute*	*Relates to Entity Type*
DAM	NAME	
	WIDTH	
	HEIGHT	
	DISCHARGE	
	CONTROL_OVER_WATER_LEVEL	WATERCOURSE
	FLOOD_CONTROL	
	HYDROELECTRIC_POWER	
	MANMADE/ARTIFICALLY_IMPROVED/NATURAL	
	EXISTING/PROPOSED	
WATERCOURSE	NAME	
	WIDTH	
	depth*	
	DIRECTION_OF_FLOW	
	WETTED_PERIMETER	
	CROSS_SECTIONAL_AREA	
	FORM_RATIO	
	FORCE_OF_FLOW	
	INTERMITTENT/PERENNIAL	
	RECREATIONAL	
	FEATURE_PRESENT	DAM
LAKE	NAME	
	WIDTH	
	AREA	
	VOLUME	
	depth*	
	SALINITY	
	SEASONAL_DEPTH	
	MANMADE/ARTIFICALLY_IMPROVED/NATURAL	
	EXISTING/PROPOSED	
	RECREATIONAL	
	WATER_SUPPLY	
	IRRIGATION	
	RESTRICTIONS	
	FEATURE_CONNECTED	DAM
	FEATURE_CONNECTED	WATERCOURSE

* Note: the attribute DEPTH was among several entries in *The American Cartographer* proposed standard that were removed in the final editing of SDTS. Therefore, it must currently be defined by the user.

Table 1: Selected Entity Types and Attributes.

Field	*Value*
ENTITY_TYPE	DAM
NAME	Orono Town Dam
WIDTH	200 meters
HEIGHT	10 meters
DISCHARGE	100 cubic meters per second
CONTROL_OVER_WATER_LEVEL	<foreign identifier>*
FLOOD_CONTROL	true
HYDROELECTRIC_POWER	true
MANMADE/ARTIFICALLY_IMPROVED/NATURAL	MANMADE
EXISTING/PROPOSED	existing

* points to the entry for Stillwater River in the table of watercourses

Table 2: An Instance of the Entity Type DAM.

3. Functional Algebras for Entity Type Definitions

Functional algebras, also called *universal algebras*, *algebraic specifications*, and *functional [programming] languages*, are a formal method of specifying the behavior of abstract data types. They are "… based on the evaluation of expressions, suitably generalized to allow complex data structures to be specified. The description 'functional' arises from their use of side-effect-free functions as the main program structuring and abstraction device." (Bailey, 1990).

A specification using functional algebra consists of three main kinds of expressions: abstract data types, functions, and axioms. Abstract data types are specified by *sort identifiers*, functions are specified by *operation identifiers*, and *axioms*, or equations, specify the behavior of the operations on the sorts. The combination of sort and operation identifiers constitutes the *signature* of the data type.

Functions are of two basic types: constructor functions and observer functions. Constructors add new instances or modify the current state of a sort. In terms of the SDTS model, we would say that constructor functions create or change *entity instances*. Observers return information about the sort without adding to it or changing its state. They return *attribute values* of entity instances. Axioms specify the results for each observer function of the operation of each constructor function.

Using Gofer (Jones, 1993) syntax, abstract data types can be specified using `data` statements. Functions are defined by the `::` symbol and axioms as equalities using the

symbol =. An example of a Gofer language specification is Kuhn's “glas spec.” The *sort identifier* introduces the abstract data type Glas with two possible states: it is a new glas or it has been filled:

```
data Glas = NewMug (Int) | Fill (Glas, Int)
```

The new glas is of type integer where the value is the size of the glas. The filled glas is of type (glas, integer) where the value of the integer is the amount of water added to the glas.

The *operation* on the glas is to take a drink. The *observer functions* include noting the size of the glas, the level of beer remaining, and whether the glas is empty or full. Note that the *constructor function* “drink” is an operation that affects the glas itself, whereas the observer functions simply return data values.

```
drink    ::  (Glas, Int) -> Glas
size     ::  Glas -> Int
level    ::  Glas -> Int
empty    ::  Glas -> Bool
full     ::  Glas -> Bool
```

The operation “drink” includes an argument of type *integer* which is the amount of water consumed in the drink. The resulting “glas” has that much less remaining to be consumed. The observer “size” keeps track of the maximum capacity of the glas expressed as an integer. The observer “level” keeps track of the amount of beer in the glas, another integer quantity. The Boolean operators “empty” and “full” are either true or false for a given state of the glas.

The *axioms* for the glas are as follows. Note that there are two axioms for each of the operations “drink,” “size,” and “level,” one for each of the two types of glas. The values of the observers “empty” and “full” are derived from values of the other observers “level” and “size.”

```
drink (NewMug (i), j)      = NewMug (i)
drink (Fill (m,i), j)
   | full (Fill (m,i))     =
                   Fill (NewMug (size (m)), size (m) - j)
   | i>j                   = Fill (m, i-j)
   | otherwise             = drink (m, j-1)
size (NewMug (i))          = i
size (Fill (m,i))          = size (m)
level (NewMug (i))         = 0
level (Fill (m,i))
  | (level (m) + i) > size (m) = size (m)
   | otherwise             = level (m) + i
empty (m)                  = level (m) == 0
full (m)                   = level (m) == size (m)
```

The first axiom indicates that there is nothing to drink from a new glas. The second shows that there are three possible outcomes of drinking from a filled glas. If the glas starts out full, its level is lowered by the amount of the drink. If the glas is not full and the amount of the drink is less than what remains in the glas, the level is again lowered by the amount of the drink. If the amount to be drunk is at least as much as the amount remaining in the glas, the result is that the amount to be drunk is lowered by one unit (until the amount to be drunk equals the amount remaining). The size of the new glas is given (3rd axiom) and does not change when the glas is filled (4th axiom). The level of the new glas is zero (5th axiom). The sixth axiom shows that the level of the glas increases by the amount it is filled, unless the amount of beer added is beyond its capacity, in which case the glas is simply filled up. The glas is empty if its level is zero (7th axiom) and full if it is filled to capacity (8th axiom).

The glas example can serve as an analogy for the reservoir created when a new dam is constructed. The following section presents a more detailed specification of a dam and its associated reservoir along a watercourse.

4. A Definition of Dam using Functional Algebra

For the purpose of this example, the abstract data types correspond to *entity types* in SDTS. Because functional algebra requires that we specify operations, the critical first step in arriving at a specification is to be clear about precisely which operations are involved. In the example of a dam, we can take the natural-language definition from SDTS and picture a series of operations as illustrated below. First, while the dam is only proposed, we have a single entity, the watercourse, with attributes of depth[2] and flow.

Cross-sectional views of this pre-dam situation are illustrated in Figure 1.

---

[2] To simplify the metrics of the problem, we assume that the gradient of the watercourse and its cross-sectional area, taken together, result in a constant value for the attribute "depth" over the segments where the dam will be built. We can then use this attribute to operationalize the variable "water level" in the definition of dam.

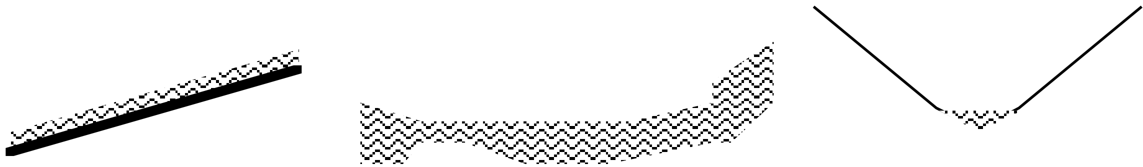

Fig. 1. WATERCOURSE

The situation can also be represented algebraically (Specification 1), with an abstract data type WATERCOURSE, whose only functions are to observe the depth and flow of the stream.

Specification 1:

```
data Watercourse           =  Stream (Int,Int)
streamDepth               ::  Watercourse -> Int
streamFlow                ::  Watercourse -> Int
streamDepth (Stream(u,v)) =  u
streamFlow (Stream(u,v))  =  v
```

4.1 Constructing a Dam

The first phrase in the definition of DAM is, "a barrier constructed across…" So, we have an operation "construct the dam." As a result of this operation, several changes occur in the objects shown in Figure 1. Of course, there is a new DAM where there was none before. The WATERCOURSE is now split into two parts that will behave differently, an upstream part and a downstream part. Also, a portion of the valley through which the WATERCOURSE flows is about to be flooded. This RESERVOIR area will become a new LAKE. Figure 2 illustrates the situation after the DAM has been built, but before the LAKE is filled.

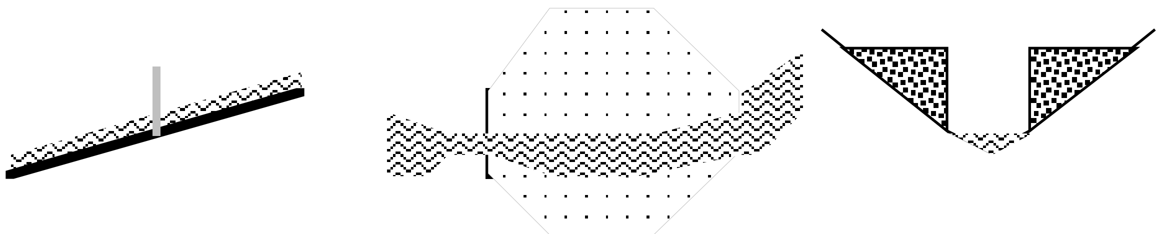

Fig. 2. New DAM

The algebraic specification now involves the entity types DAM and LAKE in addition to WATERCOURSE. The operations include constructing a dam and creating a new lake. We then observe that the new dam is (as yet) open, its height is zero, and its discharge is zero. The new lake is empty and its depth is also zero. The maximum height of the dam is set at the time of construction (Specification 2)

Specification 2:

```
data Dam         = ConstructDam (Int) | Operate (Dam,Int,Int)
data
Watercourse=Upstream(Int,Int)|Downstream(Watercourse,Int.Int)
data Lake        = NewLake (Int) | Fill (Lake, Int)

maxHeight        :: Dam -> Int
damHeight        :: Dam -> Int
damOpen          :: Dam -> Bool
streamDepth      :: Watercourse -> Int
streamFlow       :: Watercourse -> Int
lakeDepth        :: Lake -> Int
lakeEmpty        :: Lake -> Bool

maxHeight (ConstructDam (k))     = k
damHeight (ConstructDam (k))     = 0
damOpen (ConstructDam (k))       = True
streamDepth (Upstream(u,v))      = u
streamDepth (Downstream(w,u,v)) = streamDepth (Upstream (u,v))
streamFlow (Upstream(u,v))       = v
streamFlow (Downstream(w,u,v))  = streamFlow (Upstream (u,v))
lakeDepth (NewLake(m))           = 0
lakeEmpty (NewLake(m))           = True
```

4.2 Raising a Dam

The next operation is to raise the height of the dam and begin to fill the reservoir. This is illustrated in Figure 3.

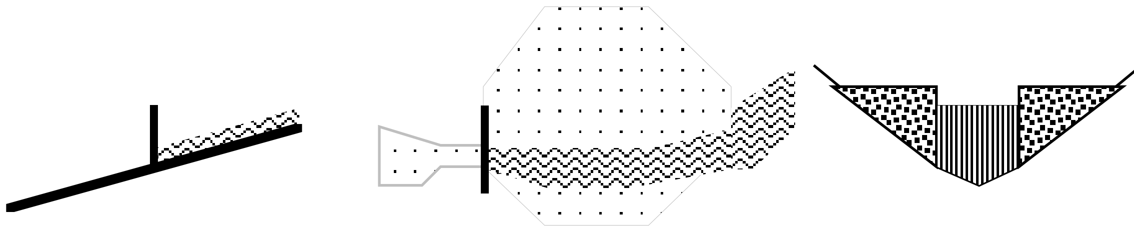

Fig. 3. Raise DAM

We add the operation of raising the dam, which results in the dam being closed and stopping the flow of water downstream. We add a condition to the operation of raising the dam that prevents the dam from being raised higher than its maximum possible height (or lower than zero). The mutual interdependence of the entity types is reflected in more complex operations and equations (Specification 3).

Specification 3:

```
data Dam         = ConstructDam (Int) | Operate (Dam,Int,Int)
data Watercourse= Upstream(Int,Int)|Downstream(Watercourse,Int,Int)
data Lake        = NewLake (Int) | Fill (Lake, Int)

damRaise     :: (Dam,Int) -> Dam
maxHeight    :: Dam -> Int
damHeight    :: Dam -> Int
discharge    :: Dam -> Int
damOpen      :: (Dam,Watercourse,Lake) -> Bool
damClose     :: (Dam,Watercourse,Lake) -> Bool
streamDepth  :: (Dam,Watercourse,Lake) -> Int
streamFlow   :: (Dam,Watercourse,Lake) -> Int
lakeDepth    :: (Dam,Watercourse,Lake) -> Int
lakeEmpty    :: (Dam,Watercourse,Lake) -> Bool

maxHeight (ConstructDam(k)) = k
maxHeight (Operate (d,i,j)) = maxHeight (d)
damHeight (ConstructDam(k)) = 0
damHeight (Operate (d,i,j)) = i
discharge (ConstructDam(k)) = 0
discharge (Operate (d,i,j)) = j

streamDepth (d,Upstream(u,v),l) = u
streamDepth (d,Downstream(w,u,v),l)
  | damClose (d,w,l) == True = 0
  | damOpen (d,w,l) == True  = streamDepth (d,w,l)
    + lakeDepth(d,w,l) - damHeight (d)
  | otherwise                = streamDepth (d,Upstream(u,v),l)
streamFlow (d,Upstream(u,v),l)  = v
streamFlow (d,Downstream(w,u,v),l)
  | damClose (d,w,l) == True = 0
  | damOpen (d,w,l) == True = streamFlow (d,w,l)
   + discharge (d)
  | otherwise               = streamFlow
(d,Upstream(u,v),l)

damOpen (ConstructDam (k),w,l) = True
damOpen (Operate(d,k),w,l)     = damHeight (d) < lakeDepth (d,w,l)

lakeEmpty (ConstructDam(d),w,l) = True
lakeDepth (ConstructDam(d),w,l) = 0

damRaise (ConstructDam (d),h)         = error
   "Cannot raise height of a dam under construction"
damRaise (Operate (d,i,j),h)
         | (h>i) && (h < maxHeight(d)) = Operate (d,h,j)
         | otherwise = error "Illegal new height for dam"
damClose (d,w,l) = damHeight (d) > lakeDepth(d,w,l)
```

4.3 Reservoir is Full

The above state of affairs continues until the reservoir fills to the level of the dam, as illustrated in Figure 4.

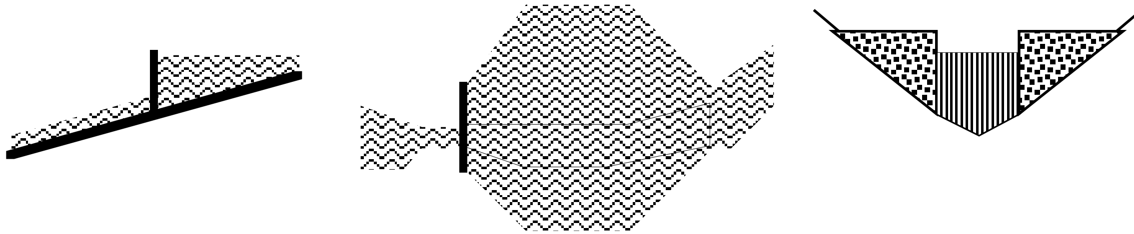

Fig. 4. LAKE Full

We add an observer function to indicate when the lake is full (Specification 4).

Specification 4:

```
lakeFull         :: (Dam,Watercourse,Lake) -> Bool
lakeFull (d,w,l) = lakeDepth (d,w,l) == damHeight (d)
```

When this occurs, the dam is neither "open" (discharging extra water into the downstream segment of the watercourse) nor "closed" (preventing any water from flowing downstream). The downstream segment of the watercourse returns to its normal upstream depth.

4.4 Lowering a Dam (discharging)

An operation that has different consequences is illustrated in Figure 5: lowering the dam.

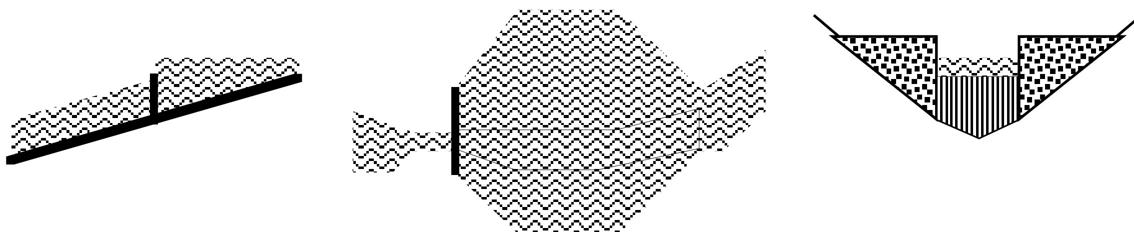

Fig. 5. Lower DAM

When this happens, there is a period of time when the height of the reservoir exceeds that of the dam. The downstream flow is increased relative to the upstream flow by the additional amount of discharge from the dam. The dam-lowering operation and its effects are shown in Specification 5.

Specification 5:

```
discharge :: (Dam, Watercourse, Lake) -> Int
discharge (ConstructDam (k),w,l) = 0
discharge (Operate (d,i,j),w,l)
  | damOpen (d,w,l) = (( lakeDepth (d,w,l) - damHeight (d)) /
   streamDepth (d,w,l)) * streamFlow (d,w,l)
damLower  :: (Dam,Int) -> Dam
damLower (ConstructDam (k),h) = error "Cannot lower a new dam"
damLower (Operate (d,i,j),h)
        | (h<i) && (h >= 0) = Operate (d,h,j)
        | otherwise = error "Illegal new height for dam"
```

As shown in Figure 6, eventually the level of the reservoir falls to the height of the dam and the system is again in equilibrium (as observed by the functions "dam not open", "dam not closed", and "lake full").

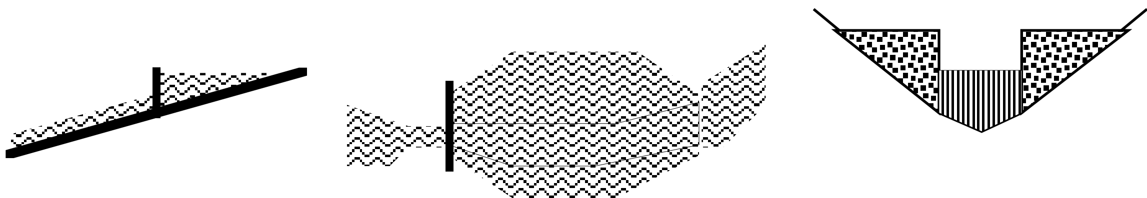

Fig. 6. LAKE FULL

This condition was specified earlier. The only difference is that there are new values of the observations for the height of the dam and the depth of the lake. A complete specification, including all the operations described above, is appended.

5. Conclusions

5.1 Summary

The natural-language definition for DAM expresses operations that can be formalized in terms of algebraic specifications for the operations (construct, raise, lower, open, close) and their effects (discharge, stream depth, stream flow, lake depth, lake empty, lake full). Such specifications are precise and capable of being expressed in any number of natural languages without ambiguity.

Functional algebra offers a formal method for refining the work begun in an informal fashion with the development of Part 2 of SDTS. The SDTS model of entity types and attributes is rigorously supported in algebraic specifications in the form of abstract data types and observer functions. The example of the dam shows that relationships among entity types can also be precisely modeled using functional algebra. In order to express

the operation of a dam and its effects, one must include the related entity types WATERCOURSE and LAKE as arguments to some of the functions of the dam.

Functional algebra goes beyond current entity-attribute-relationship models by specifying the operations of entity types. Without this methodology, the behavior of such phenomena as roads, dams, and watercourses can only be described through a natural-language definition, or modeled through a specific computer program that accompanies the data.

5.2 Discussion

Attributes are specified in terms of observer functions that are needed to describe the operations of entity types and (or) the effects of these operations on other entity types. The standard attributes defined in SDTS were obtained informally through an analogous process: by determining which attributes were necessary to distinguish one standard entity type from another, or to capture the nuance of meaning between a standard entity type and an “included term.” Functional specifications offer the possibility of refining the selection of attributes to include only those that are essential to understanding the operations of entity types.

Functional specifications also offer an advantage for transfer of data across applications. A given view of an entity type can be specified as a series of operations. For example, the effects of raising the dam may be of interest to a hydrologist and also to an ecologist, whereas its mere existence may be sufficient for the purposes of an airplane navigator. Functional specifications cannot solve the problem that data collected for one purpose may not meet other requirements, but at least they can help to clarify whether a problem exists and why.

Perhaps the most significant contribution that functional algebra can make in the area of exchange standards is to identify functionally equivalent entity types. If the signature of a given entity type in one system matches, or is mathematically equivalent, to that of an entity type in another system, these entity types are not only synonyms—in the sense of natural language—but also they are, strictly speaking, interoperable. Thus, algebraic specifications of entity types can be useful in overcoming cultural and linguistic barriers to spatial data exchange.

Algebraic specifications also present some difficulties. Although fairly easy to use for experienced programmers familiar with the method, they are not intuitively obvious to the beginner. There are, for example, three possible kinds of statements that can be used to specify abstract data types: a type statement, a data statement, or a class statement. The arguments to functions appearing on the right hand side of an axiom must be also appear on the left hand side, which in turn imposes a rigorous control over data type specifications and the form of functional specifications. We were unable to find reference materials written at a sufficiently high level to set forth the full power of functional programming languages to specify complex geographic entity types and their behaviors. The examples presented here only succeeded after many hours of trial and error. Better reference materials are needed before this approach can be widely accepted as a basis for feature definitions.

5.3 Future Work

We envision perhaps a two-stage process in the future evolution of standards for entity type definitions. Starting immediately, is it possible to review current standards and begin to develop operational specifications to represent the meanings that are implicit in the natural-language definitions of entities, attributes, and relationships. The systematic thinking that is required to express these operations algebraically should lead to immediate improvements in such areas as conciseness of definitions, selection of attributes, and standardization of concepts of relationships among entity types. In the not-too-distant future, perhaps these natural-language definitions will be completely replaced by mathematical specifications which, in turn, can be translated as desired into any given natural language.

Functional specifications can serve as a solid foundation for the future development of feature-based geographic information systems (GIS) in an Open GIS environment.

6. References

Bailey, Roger (1990). *Functional Programming with Hope*. London: Ellis Horwood Ltd.

Frank, Andrew U. (1994). “Qualitative Temporal Reasoning in GIS-ordered Time Scales.” Thomas C. Waugh and Richard G. Healey (eds.), *Advances in GIS Research.* Proceedings of the 6th International Symposium on Spatial Data Handling, Edinburgh UK, September 1994.

Frank, Andrew U. and Werner Kuhn (1995). “Specifying Open GIS with Functional Languages.” Max J. Egenhofer and John Herring (eds), *Advances of Spatial Databases*. Proceedings of the Fourth International Symposium on Large Spatial Databases, Portland ME, August 1995.

Jones, Mark P. (1993). “Gofer Functional Programming Environment,” version 2.28. Available via anonymous FTP at site ftp@dcs.glasgow.ac.uk in the directory /pub/haskell/gofer .

Kuhn, Werner (1994). “Defining Semantics for Spatial Data Transfers.” *Proceedings, Sixth International Symposium on Spatial Data Handling*. Edinburgh: International Geographical Union, 973-987.

Kuhn, Werner and Andrew U. Frank (1991). “A Formalization of Metaphors and Image-Schemas in User Interfaces.” David M. Mark and Andrew U. Frank (eds), *Cognitive and Linguistic Aspects of Geographic Space*. Proceedings of the NATO Advanced Study Institute on Cognitive and Linguistic Aspects of Geographic Space. Dordrecht: Kluwer Academic Publishers.

Mark, David M. (1993). “Toward a Theoretical Framework for Geographic Entity Types.” Andrew U. Frank and Irene Campari (eds), *Spatial Information Theory: a theoretical basis for GIS*. Proceedings of COSIT '93. Berlin: Springer-Verlag, 270-283.

Rowell, Todd J. and Max J. Egenhofer (1995). “Translating Algebraic Specifications into a Visual Programming Language.” Orono: National Center for Geographic Information and Analysis.

SDTS (1992). *Spatial Data Transfer Standard (SDTS)*. Federal Information Processing Standards Publication 173 (FIPS 173). Part 2: “Spatial Features.” U.S. Department of Commerce: U.S. Government Printing Office.

The American Cartographer, special issue (January 1988). *Digital Cartographic Data Standards*. Volume 2: “Spatial Features.”

Zemankova, Maria (1995). Personal Communication.