SPECTalk: An Object-Oriented Data Specification Language

Duane Szafron
Department of Computing Science, University of Alberta,
Edmonton, AB, T6G 2H1 CANADA

ABSTRACT

This paper describes SPECTalk, an object-oriented language for specifying data. SPECTalk uses (hierarchical) object encapsulation, instantiation, inheritance and constraints to specify both the definitions of data formats and the representation of data instances. In addition, because SPECTalk is an executable specification language, specifications can be executed to check their validity. In addition to its object-oriented features, SPECTalk also provides full support for the specification of relational databases.

The SPECTalk language rests on top of Smalltalk. In turn, SPECTalk is intended as a base layer for more specialized data specification languages. For example, SPECTalk serves as the base layer for SAIF, a specification language for geographic information systems that is being used by the government of British Columbia.

KEY WORDS: Object-oriented, Specification, Inheritance, Geographic Information Systems

1.0 Introduction

How do we specify a large collection of dependent data formats in a simple compact, readable and extensible way? How do we ensure that a data stream that is supposed to be in one of these formats is valid?

There are hundreds of existing specification languages designed for everything from program specification to protocol specification. Is a new specification language really necessary? A component of most program specification languages like Z [Pott91], Object-Z [Carr89], CLEAR [Burs81] , GYPSY [Ambl77] or VDM [Jone86] could be used for data specification, since the programs that they specify must manipulate data. Alternately, a protocol specification language like ASN.1 [Vand89] or LOTOS [Blac89] could be modified to specify data since data can be viewed as a stream of information whose format is a protocol. Finally, a database data model like GemStone [Serv89] could be used to specify data formats [Ullm88].

To understand the language requirements for data specification, it is useful to look at an example. The domain of geographic information systems (GIS) will be used in this paper, but the principles are domain independent. Our experience with data specification in geographic information systems can provide general guidelines for data specification in general. Section 2 derives requirements that any specification language must satisfy to define geographic data. Section 3 discusses the inability of existing protocol specification languages and databases to meet these requirement and argues for the use of objects and inheritance. Section 4 describes a solution to the problem of data specification by presenting the executable object-oriented data specification language, SPECTalk. The SPECTalk model is described by using examples from the GIS domain.

This is a pre-print of a copyrighted article in Technology of Object-Oriented Languages and Systems Conference 8, August, 1992, pp. 123-137.

SPECTalk has been completely implemented in Smalltalk V/PM [Smal89] and is currently being used by the British Columbia Ministry of Crown Lands. They have used it to define and implement their geographic information systems (GIS) data format language called Spatial Archive and Interchange Format (SAIF) [Spat91B]. However, SPECTalk is independent of GIS concepts.

2.0 Language Requirements for Geographic Data

There are a growing number of independent GIS, each with its own representational scheme and hardware implementation. However, there is an increasing cost associated with maintaining these isolated repositories of geographic data and with integrating information that is obtained from them. The problem is to develop a simple and powerful specification language that can be used to specify a common archival format, user-defined extensions and to transfer, integrate and validate information from these disparate sources [Spat91A].

In GIS, the data can be grouped into three categories: <u>geographic feature concepts</u> (roads, rivers, etc.), <u>geometric concepts</u> (lines, polygons, etc.) and <u>descriptive concepts</u> (names, capacities, locations, etc.). Any data specification language must be able to specify concepts from all three diverse areas. This section derives requirements for such a specification language.

2.1 Geographic Features

Two examples should serve to convey the fundamental nature of geographic feature specification. The first example describes a detailed specification of roads and trails. The second example is more general and describes the specification of a group of topographic features with varying levels of detail depending on the feature type.

Road/Trail Example

A road is described by six features: 1) divided or not, 2) hard, loose, improved, unimproved or winter, 3) a number of lanes from one to eight or possibly more than eight, 4) one-way or two-way, 5) operational, under construction or proposed and 6) elevated, not elevated or depressed.

A trail is described by three features: 1) portage, ski, snowmobile, bicycle, equestrian or pedestrian/hiking, 2) paved, boardwalk, improved or unimproved and 3) operational or under construction.

<u>A GIS specification language must be capable of specifying geographic features to a high level of detail.</u> Notice however, that there is some similarity between the specifications of road and trail. That is, they have some common features. <u>A GIS specification language must be able to abstract similar features so that they need only be specified once.</u>

General Topographic Example

A topographic feature may either be described no further, or may be further described as one of a hydrographic feature, a transportation feature, a landmark feature or a land cover feature. A hydrographic feature is either man-made, natural or neither. It may either be described no further or it may be described as one of a watercourse, a waterbody or a structure. A natural watercourse may either be described no further or may be described as a river/stream. A man-made watercourse may either be described no further or may be described as either a canal or a flume. If a watercourse is neither natural nor man-made, it may not be described further. A river/stream is either definite, intermittent, indefinite, right bank left bank, dry or nothing.

<u>A GIS specification language must be capable of specifying information to varying degrees of detail.</u> Notice that the degree of

detail sometimes depends on the specific value of a feature, like whether the feature is natural, man-made or neither. A GIS specification language must be able to specify variants based on feature values.

2.2 Geometric Concepts

A handful of geometric concepts are used extensively in GIS. These concepts include points, lines, arcs and polygons. However, since these concepts are used in a geographic context to represent the geometry of features, the concept of topology plays an important part. Consider a river system consisting of a collection of arcs that represent the position of rivers. A specification of a river system must include topological information describing which arcs are upstream from which other arcs. For example, consider the river system whose position is represented by Figure 1. It consists of four arcs labelled: A, B, C and D. A specification language must be able to indicate that Arcs A and B are upstream of arcs D and E, while arcs C and D are upstream of arc E.

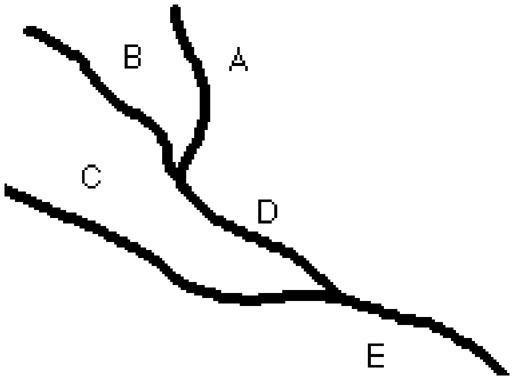

Figure 1. Topology is important

A GIS specification language must be able to represent the topological relationships between geometric concepts.

2.3 Descriptive Concepts

Database relations provide a general mechanism for representing information and are often used for defining descriptive concepts in GIS. For example, a relation can describe the mineral deposits shown in Figure 2. The first two columns represent attribute names and domains for the features. Each of the other columns (tuples) represents a single mineral deposit with values for the features (attributes).

Mineral Deposits

Attributes	Domains	Values	Values
minfileNo	String06	'82M988'	'82M999'
latitude	String02	'56'	'58'
longitude	String02	'65'	'75'
producer	Boolean	true	false
pastProd	Boolean	false	false

Figure 2. A relation table for GIS

A GIS specification language must be capable of representing database relations. However, in GIS applications, it would be useful to let the attribute values themselves be features that are described in other relations.

Mineral Deposits

Attributes	Domains	Values	Values
minfileNo	String06	'82M988'	'82M999'
latitude	String02	'56'	'58'
longitude	String02	'65'	'75'
producer	Boolean	true	false
pastProd	Boolean	false	false
names	Names	n1	n2

Names (n1)

Attributes	Domains	Values	Values
name	String14	'Wyndy Craggy'	'Fictional Name'

Names (n2)

Attributes	Domains	Values
name	String14	'Mother Load'

Figure 3. Complex relation tables for GIS

A relation that contains other relations is called a complex relation (they are called nested relations in database terminology but are the pre-cursors of complex objects in object-oriented databases). For example, Figure 3 shows three relations. A revised form of the Mineral Deposits relation is shown which contains a Names relation for each of its two tuples.

While it is true that such complex relations can be rewritten in terms of simple relations, their abundance in GIS makes such rewriting inconvenient. Although support for complex relations is not essential for a GIS specification language it is recommended. <u>Therefore, a GIS specification language should be able to represent complex relations.</u>

2.4 Definition versus Instantiation

To this point, there are seven requirements for GIS specification languages together with one recommendation. Before summarizing these requirements it is useful to take a closer look at the specification process to identify several more requirements. The specification process can be decomposed into two parts: definition of structures and instantiation (or creation) of structures.

For example, a road is first defined by specifying its features. Here is an informal definition of a road:

```
Road: (divided or not divided),
(hard, loose, improved, unimproved
or winter), (lanes from one to eight
or possibly more than eight) , (one-
way or two-way), (operational, under
construction or proposed),
(elevated, not elevated or
depressed).
```

Several roads may be instantiated by assigning particular values for each feature in the specification. For example, here are the informal instantiations of two particular roads:

```
road1: (divided), (improved), (four
lanes), (two-way), (under
construction), (depressed)

road2: (not divided), (hard), (two
lanes), (two-way), (operational),
(not elevated)
```

<u>A GIS specification language must be able to specify definitions and specify instantiations.</u>

2.5 Validation

During instantiation, it is possible to list illegal values for features, to add non-existing features or to omit required features. Therefore, each instantiation should be validated to ensure that it satisfies the definition. For example, here are invalid informal instantiations of three roads:

```
road3: (separated), (hard) , (two
lanes) , (two-way), (operational),
(not elevated)

road4: (improved) , (four lanes) ,
(two-way), (under construction),
(depressed)

road5: (Victoria) (Nanaimo) (not
divided), (hard) , (two lanes) ,
(two-way), (operational), (not
elevated)
```

Road3 is invalid since 'separated' is not a valid feature value. Road4 is invalid since the first feature, (divided or not divided) is missing. Road5 in invalid since it has two extra features, 'Victoria' and 'Nanaimo'.

This kind of validation can only be performed if the specification language is executable. That is, if both the definitions and instantiations are executed as language commands so that the instantiations can be compared to the definitions. Although it can be argued that validation is not necessary, it is certainly recommended. <u>Therefore, a GIS specification language should be able to validate instantiations against definitions and report errors.</u>

2.6 Syntactic versus Semantic Constraints

During the definition phase of specification it is often useful to specify constraints. For example, the values of features are constrained by listing the valid values. This kind of constraint is called a syntactic constraint since during instantiation, a feature value can simply be compared to the allowable values to determine correctness. <u>A GIS specification language should support syntactic constraints.</u>

However, other constraints are often useful that are not syntactic. For example, given a definition of Line as a list of Line Segments, it is possible to define a Connected Line as a Line subject to the constraint that the end point of each of its line segments is the start point of its next line segment. Such a constraint is called a semantic constraint since during instantiation, a computation must be performed to determine whether the constraint is satisfied.

It is not absolutely necessary for a GIS specification language to support semantic constraints. <u>However, a GIS specification language that supports semantic constraints provides a powerful abstraction mechanism.</u>

2.7 Summary of Requirements for a GIS Data Specification Language

A GIS specification language:

#1 must be able to represent concepts from three categories: geographic features, cartographic concepts and database concepts.

#2 must be capable of specifying geographic features to a high level of detail.

#3 must be able to abstract similar features so that they need only be specified once.

#4 must be capable of specifying information to varying degrees of detail.

#5 must be able to specify variants based on feature values.

#6 must be able to represent the topological relationships between cartographic concepts.

#7 must be capable of representing database relations.

#8 should be able to represent complex database relations.

#9 must be able to specify definitions and specify instantiations.

#10 should be able to validate instantiations against definitions.

#11 should support syntactic constraints.

#12 should support semantic constraints.

3.0 Choosing A Specification Language

There are three approaches to choosing a GIS specification language. The first approach is to use an existing (protocol or programming) specification language for GIS specifications. The second approach is to use an existing database language for GIS specifications. The third approach is to create a new specification language that meets the requirements and recommendations for a GIS specification language.

3.1 Using Existing Specification Languages

A protocol specification language like ASN.1 can be used to specify Geographic features and Cartographic concepts. ASN.1 can be evaluated as a GIS specification language using the requirements that were listed in section 2 of this paper. It meets or partially meets the requirements: #1 (geographic and cartographic concepts only), #2, #4 and #9.

If an interpreter is constructed to execute the ASN.1 specifications then recommendations #10 and #11 can be met. However, even with an executing interpreter, ASN.1 fails to meet the requirements #1 (database concepts), #3, #5, #6, #7 and the recommendations #8 and #12.

Other existing protocol and program specification languages also fail to meet the majority of requirements and recommendations listed in this report.

3.2 Using Databases

It is possible to take an existing database system and use it as a GIS specification language. This approach makes it easy to satisfy the database requirements. It is also possible to satisfy several other requirements as well. However this approach is inadequate or very tedious for several of the requirements #3, #5 and #9, and recommendations #8 and #12.

3.3 New Specification Languages

Since existing languages cannot meet the requirements and recommendations for a GIS specification language, a new language should be constructed. However, unless a new technology is used, it is doubtful that the problems with existing languages can be overcome. For example, constructing a new executable specification language based on ASN.1 with additional facilities to meet the requirements of variants, better abstraction, database facilities and semantic constraints would change the character of ASN.1 so much that it would be unrecognizable. In addition, it would be difficult to add such disparate capabilities and maintain the simple model of ASN.1

3.4 Using Objects

An executable object-oriented GIS specification language has several inherent advantages over traditional specification languages and databases. Besides the component based capabilities (hasA) provided by traditional languages, object-oriented languages provide a powerful abstraction mechanism called inheritance (isA) that can easily be used to directly meet several of the requirements #2, #3, #4, #5 and #9. In addition, the message passing paradigm can be combined with inheritance to meet the additional recommendations #10, #11 and #12.

The only remaining challenge that must be faced in using an executable object-oriented specification language is the provision for database support. This challenge has been faced and met by the SPECTalk layer in the layered specification system described in the next section.

3.5 A Layered Object-Oriented Specification System

To meet the requirements of a GIS data specification language as outlined in this paper, an object-oriented specification system has been constructed. However, to ensure domain independence, the system consists of four layers as shown in Figure 4. The first (base) layer is Smalltalk. This layer provides an object-oriented language complete with semantics for objects, classes and messages and a syntax for expressions, message sending and methods. It also provides several basic pre-defined classes: Character, Boolean, String, Symbol, SmallInteger (Integer), Float, Array, Object and UndefinedObject that can be used in all other layers.

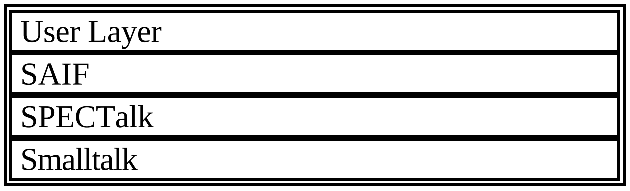

Figure 4. A layered object-oriented specification system

The second layer is SPECTalk. This layer provides static typing, constraints and the syntax for definitions and instantiation of objects and classes. It provides the class SPECObject, the common superclass for all other SPECTalk, and Application specific class definitions and common messages that can be sent to all SPECTalk objects. It also provides the class Enumeration, the superclass of user defined scalar definitions and associated messages. In addition, it provides seven classes: Relation, BinaryRelation, AntiSymmetricRelation, SymmetricRelation, TransitiveRelation, List and Tuple that serve as a basis for database concepts and many powerful messages that can be applied to objects of this type, including messages that implement the complete relational algebra. SPECTalk is completely independent of GIS concepts.

The third layer is SAIF. This layer provides a set of standard definitions of geographic and cartographic objects and messages that can be sent to them [Spat91B]. The fourth layer is the user layer. This layer consists of all user defined objects and messages. The remainder of this paper gives a detailed description of SPECTalk.

4.0 The SPECTalk Model

SPECTalk is an executable object-oriented specification language that can be viewed as a specification layer that resides on top of a Smalltalk/V object-oriented programming layer. A user layer of domain-specific definitions can be created as a third layer (like SAIF was). Alternately, two layers may be added where the third layer is a set of general domain-specific definitions and the fourth layer is a set of specialized domain-specific definitions.

Smalltalk was used for two reasons. First, it is a "pure" object oriented language so it produced a specification language where every construct was a first-class object. Second, it was very easy to produce a working system in a short period of time, complete with SPECTalk specific browsers that were used to construct SAIF. Despite these reasons for choosing Smalltalk, there is no fundamental reason why a different object-oriented language could not be used instead.

Following the requirements, the specification task is decomposed into two phases, definition and instantiation. Definitions are realized by defining a new SPECTalk class (domain) for each definition. For example, Road would be a class. The classes serve as templates for the objects that are to be created during the instantiation phase. Instantiation is realized by sending specific creation messages to the classes so that instance objects are returned. In addition, messages can also be sent to these instantiated objects to solicit information from them or to modify them.

There are three groups of classes in SPECTalk. The first group consists of those classes that are pre-defined by Smalltalk and used in SPECTalk. They are called Smalltalk classes: Character, Boolean, String, Symbol, SmallInteger, Float, Array and UndefinedObject.

The second group consists of those classes that are not pre-defined by Smalltalk, but are pre-defined by SPECTalk. They are called SPECTalk classes: SPECObject, Enumeration, Tuple, Relation, BinaryRelation, AntiSymmetricRelation, SymmetricRelation, TransitiveRelation and List.

The third group consists of those classes that are defined using SPECTalk but are not pre-defined. They are called Application classes. SPECTalk commands are used to define Application classes as sub-classes of SPECTalk classes.

4.1 Smalltalk Classes

SPECTalk uses eight pre-defined Smalltalk classes that have literal representations. These classes and some example literals are shown in Figure 5. These classes are not part of the SPECTalk Class hierarchy since they are defined in Smalltalk. Notice that symbols cannot contain blanks and that arrays can contain elements from different classes.

Class	Example Literals
SmallInteger	5 35231 -18
Float	3.45 -2.03 0.2
Character	$a $A $5 $?
String	'82M998'
Symbol	#improved
Array	#('82M998' 56 65 true false)
Boolean	true false
UndefinedObject	nil

Figure 5. Pre-defined Smalltalk classes with literal instances

4.2 SPECTalk and Application Class Structure

All SPECTalk classes are organized into a tree structure that provides inheritance of behavior as shown in Figure 6. Every new class is defined as a leaf node in this tree. Of course a leaf node will become a non-leaf (interior) node if a new class is subsequently defined as a subclass of it.

This tree defines an 'isA' hierarchy. For example, every instance of Relation is a SPECObject and every instance of BinaryRelation is a Relation. This means that every message that can be sent to an instance of SPECObject can also be sent to any other SPECTalk object. However, as one moves down the tree, new messages can be added that are not understood by objects higher in the tree. For example, there are messages that instances of Relation understand that are not understood by instances of SPECObject. In fact, all SPECTalk classes are abstract superclasses for domain-specific Application classes defined by users.

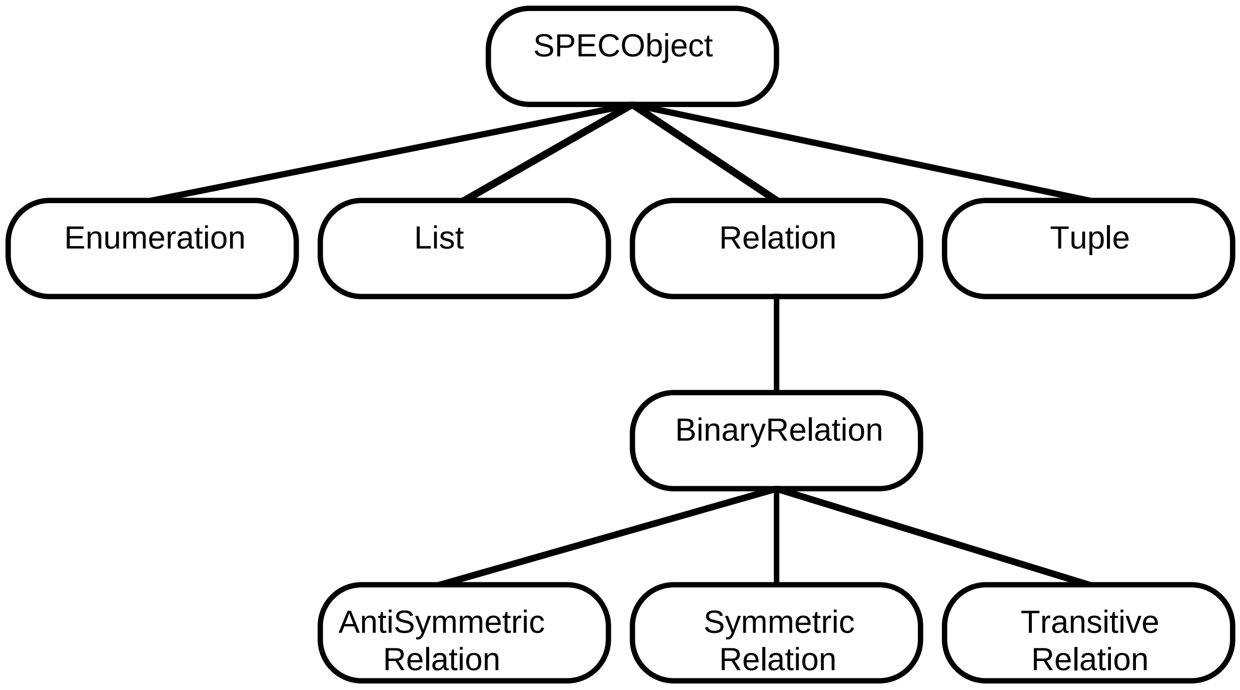

Figure 6. The Class Hierarchy of SPECTalk

4.3 SPECObjects

SPECObject is the root class for all SPECTalk classes. That is, all SPECTalk classes (and therefore all Application classes as well) are sub-classes of it. SPECObject defines the behavior that is common to all SPECTalk and Application classes:

#1 Every SPECTalk and Application class has a comment that describes its purpose. Application class comments can be edited.

#2 Every SPECTalk class has a set of instance methods and a set of class methods. Every Application class has a set of instance methods. Application methods may be defined and undefined.

#3 Every Application class has a set of constraints that each instance of that class must satisfy. Each constraint is represented by an instance message. An instance of a class satisfies a constraint if it responds to the message by answering true. An instance violates the constraint if it answers any other value than true. Whenever an instance is created or modified, its constraints are automatically checked and all violations are reported. Each Application class inherits the constraints of all of its superclasses. Each SPECTalk or Application class can be used to define an Application sub-class of itself with additional constraints. SPECTalk classes have no constraints.

#4 Each SPECTalk or Application class can be used to un-define (delete the definition of) any of its Application sub-classes.

4.4 Enumerations

The class Enumeration is the root class for all Application Enumeration classes. That is, all Enumeration classes are sub-classes of it. Enumeration defines the behavior that is common to all Enumeration classes:

#1 Every Enumeration class defines a set of possible values that are instances of class Symbol. Each instance can have one of the specified values or a special value denoted nil that represents an unspecified value. The value nil is not an instance of class Symbol. It is the only instance of the class UndefinedObject. However, to avoid confusion, the symbol #nil is not allowed as a valid value for Enumeration classes.

For example, an Enumeration class called Surface may be defined with the values: #improved, and #unimproved. Each instance of Surface has one of the values: #improved, #unimproved or the unspecified value nil.

#2 Each Enumeration class inherits the values of all of its Enumeration superclasses, although the class Enumeration itself has no values.

For example, an Enumeration class called RoadSurface may be defined as a sub-class of Surface. The additional values: #hard, #loose and #winter may be specified in addition to the inherited values #improved and #unimproved.

#3 Each Enumeration class has a default value that is used when an instance is created without specifying a value. The default value is specified when the Enumeration class is defined. If no default value is specified, then the new class inherits the default value from its superclass. The default value of the class Enumeration is nil.

#4 Instances of Enumeration classes can be created in several ways. The message #create: can be used to create a new instance of an Enumeration class whose value is the message argument. The message #create can be used to create a new instance of an Enumeration class with the default value. The message #nil can be used to create a new instance of an Enumeration class with a nil value. In addition, when an Enumeration class is created, a message is created for each of the new values. The names of these messages are the values themselves. Each message creates a new instance whose value is the name of the message.

For example, when the Enumeration class Surface is created, two messages with names: #improved and #unimproved are created automatically. An instance of Surface with the value #improved can be created by sending the message #improved to the class Surface.

4.5 Tuples

The class Tuple is the root class for all Application Tuple classes. That is, all Tuple classes are sub-classes of it. Tuple defines the behavior that is common to all Tuple classes:

#1 Every Tuple class defines a list of attributes. The attributes are the names of components that instances of the Tuple class contain. Each attribute is a Symbol.

For example, a Tuple subclass called MineralDeposit may be defined with the attributes: #minfileNo, #latitude, #longitude, #producer and #pastProducer. Each instance of MineralDeposit would contain a number of tuples. Each tuple would contain five components, with names: #minfileNo, #latitude, #longitude, #producer and #pastProducer, in order.

#2 Associated with each attribute is a domain. Each domain must be an existing Smalltalk, SPECTalk or Application class. The components of each instance of each Tuple class must be elements from the appropriate domain.

For example, the Tuple subclass MineralDeposit could have the domains String, SmallInteger, SmallInteger, Boolean and Boolean, in order. The components: '82M998', 56, 65, true and false would be a valid tuple components while the components: 998, 56, 65, true and false would not.

#3 Each Tuple class inherits the attributes and domains of all of its Tuple superclasses, although the class Tuple has no attributes or domains.

For example, a class called PrivateMineralDeposit may be defined as a sub-class of MineralDeposit. When it is defined, the additional attribute #owner with domain SmallInteger may be specified in addition to the inherited attributes #minfileNo, #latitude, #longitude, #producer and #pastProducer.

#4 When a Tuple class is created an instance message is created for each new attribute and the selectors for these messages are the attributes themselves. When an attribute message is sent to an instance of the tuple, the value of that attribute is returned.

#5 When a Tuple class is created another instance message is also created for each attribute. The selectors for these messages are the attributes followed by a colon. When such a message is sent to an instance of the tuple, with a value as an argument, the value of the attribute in the tuple is set to the argument of the message.

For example, if the message #latitude is sent to the instance of MineralDeposit:

```
MineralDeposit ('82M998' 56 65 true
false)
```

then the value 56 would be returned. If the message #latitude:55 was sent to this instance then the instance would change to:

```
MineralDeposit ('82M998' 55 65 true
false).
```

#6 Instances of Tuple classes can be created in three ways. The message #create: can be used to create a new instance of a Tuple class. The components of the argument array are used to initialize the tuple components. There are two ways the argument may be used to initialize the tuple. The array may contain component values, in order, that are used to set the attributes. Alternately, the array may contain some attribute names where each attribute name is followed by the character < and a value. If this format is used, then only the attributes whose names appear are set to the corresponding values and the other attributes are set to nil (UndefinedObject).

The third way to create a tuple uses a different creation message. Whenever a new Tuple class is defined, a creation message is created whose selectors are the attributes of the tuple followed by colons. When this message is sent to the Tuple class, a new instance is created with components sent to the arguments of the message.

For example, the following creation messages could be sent to MineralDeposit to create the instances shown. SPECTalk output is shown in **bold** font.

```
MineralDeposits create: #('82M998'
56 65 true false)
   MineralDeposit('82M998'  56
65  true  false)
```

```
MineralDeposits create:
#(latitude<56 longitude<65
minfileNo<'82M998' producer<true
pastProducer<false)
   MineralDeposit('82M998'  56
65  true  false)
```

```
MineralDeposits create:
#(latitude<56 minfileNo<'82M998'
producer<true)
   MineralDeposit('82M998'  56
nil  true  nil)
```

```
MineralDeposits minfileNo: #'82M998'
latitude: 56 longitude: 65 producer:
true pastProducer: false
   MineralDeposit('82M998'  56
65  true  false)
```

4.6 Relations

The class Relation is the root class for all Application Relation classes. That is, all Relation classes are sub-classes of it. Relation defines the behavior that is common to all Relation classes:

#1 Every Relation class has a domain that is an existing Tuple class. An instance of a Relation class is a set of tuples from the Tuple domain class.

For example, a Relation class called MineralDeposits may be defined with the domain MineralDeposit (defined in the last section). Each instance of MineralDeposits would contain a number of tuples. Each tuple would be an instance of MineralDeposit.

#2 Furthermore, the domain must be a subclass of the domain of the Relation class which is the superclass of the Relation class being defined. The root class, Relation has as its domain the abstract class Tuple.

For example, a subclass of the Relation class MineralDeposits, called PrivateMineralDeposits, may be defined with the domain PrivateMineralDeposits (defined in the last section) since PrivateMineralDeposit is a Tuple subclass of MineralDeposit.

#3 Each Relation class has a list of keys. Each key is a set of attributes from the attributes from the domain Tuple. Each tuple added to a relation must be unique on all keys. Keys are not inherited from superclasses and if no key is specified then the default key consists of the set of all attributes.

For example, if the Relation class MineralDeposits had two keys (minfileNo) and (latitude longitude), and an instance contained the tuples:

```
{MineralDeposit('82M998' 56 65 true
false)
MineralDeposit('83M515' 58 64 false
true)
MineralDeposit('84M612' 56 64 true
false)}
```

then it would be invalid to add any tuple with minfileNo value '82M998', '83M515' or '84M612' because of the minfileNo key. It would also be invalid to add any tuple with latitude 56 and longitude 65 or with latitude 58 and longitude 63 or with latitude 56 and longitude 64 because of the (latitude longitude) key. On the other hand, if no key was specified when the MineralDeposits class was defined then there would be a single key: (minfileNo latitude longitude producer pastProducer).

#4 For each relation class, the first key specified is called the primary key. The message #at: can be used to access the tuple in any Relation whose key value is the argument to the message.

For example, if the key (minfileNo) in the MineralDeposits Relation class is the primary key and an instance contained the tuples:

```
{MineralDeposit('82M998' 56 65 true
false)
MineralDeposit('83M515' 58 64 false
true)
MineralDeposit('84M612' 56 64 true
false)}
```

this instance would respond to the message, at:'083M 515' by returning the tuple:

```
MineralDeposit('83M515'  58  64
false  true).
```

#5 Instances of Relation classes can be created in several ways. The message #create can be used to create a new instance of a Relation class that contains no tuples. The message #create: can be used to create a new instance of a Relation class containing tuples specified by the argument. The message #with: can be used to create a new Relation containing a single tuple specified by the argument.

For example, the following creation messages could be sent to MineralDeposits to create the instances shown:

```
MineralDeposits create
   MineralDeposits{}

MineralDeposits create:#(('82M998'
56 65 false false) ('83M515' 58 64
false true))
   MineralDeposits{('82M998'  56
65  false  false)  ('83M515'  58
64  false  true)}

MineralDeposits with:#('82M998' 56
65 false false)
   MineralDeposits  {('82M998'
56  65  false  false)}
```

The standard relational operators are defined for instances of class Relation and any defined subclasses.

4.7 Lists

The class List is the root class for all Application List classes. That is, all List classes are sub-classes of it. List defines the behavior that is common to all List classes:

#1 Every List class has a domain that is an existing Smalltalk, SPECTalk or Application class. An instance of a List class is a list of values from the domain class. Furthermore, the domain must be a subclass of the domain of the List class which is the superclass of the List class being defined. The root class, List has as its domain the class Object.

For example, a List class called StringList may be defined with domain String.

#2 All instances of List classes understand the message #at: which returns the component corresponding to the index. In addition, all instances of List classes understand the message, #at:put: that replaces the component whose index is the first argument, by the value that is the second component.

For example, given a StringList with components

```
('082M 998' '083M 515' '084M 612')
```

the message at:2 would return the value '083M 515' and the message at:2 put:'083M 677' would change the components of the instance to

```
('082M 998' '083M 677' '084M 612').
```

4.8 Complex Relations

In SPECTalk, tuple domains can be tuple classes. That is, SPECTalk supports complex relations. Here is an example definition of a Tuple class called LineSegment that has two attributes, each of which is an instance of the Tuple class Point2D. Since LineSegment is a direct subclass of Tuple, it inherits no other attributes or domains. LineSegment is a complex relation. LineSegment is one of the fundamental building blocks that has been defined in SAIF for representing cartographic concepts in GIS.

```
Tuple
   subclass:   #LineSegment
   attributes: #(start end)
   domains: #(Point2D Point2D)
```

Here is an example creation message that creates a LineSegment and binds the temporary name s to it. The creation message is followed by a message that returns the start component of the line segment. In both cases the display representation follows the messages.

```
s := LineSegment create:#((1.0 2.0)
(3.0 4.0))
   LineSegment(Point2D(1.0  2.0)
Point2D(3.0  4.0)

s start
   Point2D(1.0  2.0)
```

4.9 User-Defined Messages

Some messages are pre-defined for the SPECTalk classes and are inherited by Application subclasses. Some messages are automatically created when a new Application class is created. For example, when a Tuple class is created, messages with the names of the attributes are automatically created. However, in addition to these messages, users can define their own messages that can be sent to instances of Application classes. Each user-defined message must be defined by a method. Smalltalk syntax is used for the methods and the full power of Smalltalk is available in writing the methods.

For example, a method can be defined to return the length of any instance of LineSegment:

```
LineSegment method:
   'length
"Answer the distance between my
endpoints."
^((self end x - self start x)
squared + (self end y - self start
y) squared) sqrt'
```

Here is an example that illustrates how this message can be used. An instance of LineSegment is created and then the length message is sent to this instance.

```
s := LineSegment create:#((1.0 2.0)
(3.0 4.0)).

s length
      2.82843
```

4.10 Constraints

As was described in the SPECObject section of this paper, every SPECTalk and Application class can have a set of constraints. A constraint is a user defined message that is used in a class definition. For example, consider the following message that is defined for instances of class Line.

```
Line
   method: 'connected
"Answer true if my line segments are
connected sequentially.  That is,
the end point of each of my line
segments is the start point of my
next line segment."
(1 to: self size - 1)
   do:[:index|
      ((self at:index) end ~= (self
at:index+1) start)
         ifTrue:[^false]].
^true'
```

This method can be used as a constraint in defining a subclass of Line called ConnectedLine. Whenever a new instance of ConnectedLine is created, the constraint, connected, will be applied to the new instance and an error will be reported if the new instance does not return the value true. Here is the definition of the new class:

```
Line
   subclass:#ConnectedLine
   constraints:#(connected)
```

For example, the following creation message violates the constraint:

```
ConnectedLine create:#(((1.0 2.0)
(3.0 4.0)) ((5.0 6.0) (7.0 8.0)))
   ERROR  notconnected
```

5.0 Conclusion

The SPECTalk language is an object-oriented language for data specification. It provides mechanisms for defining formats (classes or intensions) and data instances (objects or extensions). Since SPECTalk is implemented in Smalltalk, user specifications are executable so they can be validated. SPECTalk also provides a powerful mechanism for defining operations on data and these operations can be used to define constraints. The constraints can then be used in other data format specifications. SPECTalk uses inheritance not only to abstract common data formats but to abstract common operations and constraints as well so that true responsibility-driven designs can be achieved [Wirf90]. All three kinds of inheritance reduce the size and complexity of data format specifications.

SPECTalk is a domain-independent data specification language that includes classes that support the entire relational algebra. SPECTalk can be used to define a domain-specific class library that can be used as the basis for domain-specific data specification. For example, SPECTalk currently serves as the basis for SAIF, a language for specifying GIS data that is being used by the government of British Columbia. SAIF includes more than 150 GIS specific subclasses of the SPECTalk classes

Acknowledgements

This research was supported by Research Contracts 89-067 and 89-024/2 from the British Columbia Ministry of Crown Lands. Thanks to M. Sondheim, R. Menes and G. Lowe for many insightful discussions during this research and to M. Tamer Özsu for many helpful comments concerning the manuscript.

References

[Ambl77] Ambler, L., et. al., "GYPSY: A Language for Specification and Implementation of Verifiable Programs, Proceedings of the ACM Conference on Language Design for Reliable Software, SIGPLAN Notices, 12 (3), pp 1 - 10, 1977.

[Blac89] Black, S., "Objects and LOTOS", Proceedings FORTE '89 Second International Conference on Formal Description Techniques", pp 285-297, 1989.

[Burs81] Burstall, R. and J. Goguen, "An Informal Introduction to Specification using CLEAR", *The Correctness Problem in Computer Science*, ed. R. Boyer and J. Stothers Moore, Academic Press, pp 185-213, 1981.

[Carr89] Carrington, D., "Object Z: An object-oriented extension to Z", Proceedings FORTE '89 Second International Conference on Formal Description Techniques, pp 401-420, 1989.

[Jone86] Jones, C., *Systematic Software Development using VDM*, Prentice-Hall, 1986.

[Pott91] Potter, B., J. Sinclair and D. Till, *An Introduction to Formal Specification and Z*, Prentice-Hall, 1991.

[Serv89] Servio Logic, *Programming in OPAL*, Servio Logic Development Corporation, Beaverton Oregon, 1989.

[Smal89] *Smalltalk/VPM Object-Oriented Programming System: Tutorial and Programming Handbook*, Digitalk, 1989.

[Spat91A] "Spatial Archive and Interchange Format: Overview of SAIF", Surveys and Resource Mapping Branch, Ministry of Crown Lands, Release 1.0, January 1991.

[Spat91B] "Spatial Archive and Interchange Format: SAIF Interchange Specification", Surveys and Resource Mapping Branch, Ministry of Crown Lands, Release 1.0, January 1991.

[Ullm88] Ullman, J.D. *Principles of Database and Knowledge-base Systems*, Vol 1, Computer Science Press, 1988.

[Vand89] Van de Bergt, S.P. and P.A.J. Tilanus, "Attributed ASN.1", Proceedings FORTE '89 Second International Conference on Formal Description Techniques", pp 298-310, 1989.

[Wirf90] Wirfs-Brock, R., B. Wilkerson and L. Wiener, *Designing Object-Oriented Software*, Prentice Hall, 1990.