Formal Information Modelling for Standardisation in the Spatial Domain

Stephan Winter
Institute for Geoinformation
Vienna University of Technology
winter@geoinfo.tuwien.ac.at

Silvia Nittel
Data Mining Lab, Computer Science Dept.
University of California, Los Angeles
silvia@cs.ucla.edu

Abstract

This paper presents the results and experiences of applying a *formal* tool to writing specifications for software in the spatial domain. We take a specification from a standards organization to demonstrate the properties of this approach, and we compare our result with the semi-formal specification of the standards organization. The expected advantages — formal semantics of specified interfaces, total abstraction from implementation, executable prototype, and extendable algebraic structure — can be verified. However negative experiences are reported also: pure abstract code turns out to appear overloaded and difficult to read (by humans). With this work we contribute to the foundations of standardisation efforts in the spatial domain, preserving an object oriented modeling approach.

1 Introduction

This paper presents the results and experiences applying a *formal* tool to writing specifications for software in the spatial domain. Current state of the art writing such specifications is utilizing UML (Unified Modelling Language, Booch et al., 1999; OMG, 2000). UML is not formal in its semantics, and remains to be descriptive in specifying behavior of methods. This paper takes a published specification from a standards organization, translates it into a formal specification, and compares the two. The selected example is the gridded coverage, a specification taken from the bookshelf of the OpenGIS Consortium, OGC. Neither the topic is critical for this paper, nor is the organization, because our interest lies in design methodologies for standards. Standards organizations like the International Standards Organization ISO or OGC apply conceptually the same methodology in designing spatial standards (Berre et al., 2000).

1.1 What is an abstract specification?

Software specifications *are* abstract in a sense that they describe a domain — in the scope of the paper: the geographic domain, especially fields. They are called *abstract* specifications as long as they do not relate to any technology, platform, or other implementation specific issues, like data structures, or programming language constructs. Abstract specifications are part of the software engineering process; they are written by computer scientists. Basically abstract specifications represent a *domain ontology*, defining the features of the domain and their behavior in a formal language (Gruber, 1993; Guarino, 1998). For domain experts — here: geographers — abstract specifications may be difficult to read.

However, the domain experts have to check and guarantee that an abstract specification is correct, which means it describes the domain in a way domain experts perceive it. Checking a specification for correctness is a process that needs extensive communication between the authors and the domain experts, and tools supporting this communication. The most popular tool is based on a graphical interface. Graphics appeals intuitively, which gives reason for the popularity of such a tool. Nevertheless understanding graphics heavily depends on interpretation, which includes the danger of miscommunication. Our approach will be based on a language with a formal semantics. This language allows to develop *executable* specifications, thus the communication between authors and domain experts consists of discussing test cases and their behavior, instead of reading code or interpreting graphics.

1.2 Why are abstract specifications important?

Geographic information systems handle spatial features in very different ways. If systems shall co-operate with each other they need a language to do so. This language abstracts from implementation issues. Such a language is provided by *standardized interfaces* that vendors can add to their systems. Standardized interfaces can be utilized by other systems without knowledge of proprietary formats. Each system translates then the standardized structures and functionality into internal structures and methods. Such interfaces are defined by organisations like OGC or ISO by providing abstract specifications and standardized ways to implement them. Thus the bigger goal of standards is interoperability between systems, allowing data access across platforms and communication between networked value-adding services.

1.3 What is the contribution of this paper?

The motivation for this paper is twofold:

- The lack of formalized semantics of spatial datasets and operations is considered as the main handicap in interoperability of spatially enabled services. Our approach overcomes some aspects of this limitation.
- The lack of specification tools that guarantee correctness and consistency raises costs and efforts in maintenance of standards and hinders standards organizations from progress. Our approach should be cheaper in maintenance by providing executable specifications and guaranteed consistency through the specification life-cycle.

In fact it is the challenge for standards organizations to define open interfaces that can be proven to be consistent, complete, and can be implemented provable correct (for compliance testing). Algebraic specifications help (Guttag and Horning, 1978; Lin, 1998; Frank, 1999). Languages supporting to write algebraic specifications are functional programming languages, which became mature in the last decade, among them Haskell (Peyton Jones et al., 1999). Functional programming languages provide a formal syntax, code specifies *behavior* (the language atoms are functions) and can be executed (for testing correctness). Thus they are proposed for the use in standards for several years now (Kuhn, 1994, 1997; Frank and Kuhn, 1999). However, the idea was not recognized by standards organizations, probably for reasons of existing skills, apparent comprehensability of a visual interface, and popularity of other tools — UML is the dominating modeling language in software industry (Kobryn, 1999).

For that reason our research interest was a proof of practicability for writing a standard in the functional programming language Haskell, and an investigation how far abstraction can be done following this way. The hypothesis was the expectation that this approach describes the semantic aspect of interfaces (behavior) formally, can be communicated easily, can be checked for type consistency, can be tested for completeness and correctness, and separates clearly between an abstract and an implementation specification. We report our results and experiences here. The expected advantages could be verified. However negative experiences are reported also: pure abstract code turns out to appear overloaded and difficult to read by humans. With this work we contribute to the foundations of standardisation efforts in the spatial domain, preserving an object oriented modeling approach.

The paper is structured as follows. Section 2 presents the problem of writing abstract specifications in the spatial domain, and it summarizes previous work. Then our running example is presented in Section 3: the OpenGIS coverage specification. Section 4 gives an overview of algebraic specifications in functional programming languages, which is the tool that is applied in Section 5 for a formal version of the OpenGIS coverage specification. In Section 6 we investigate the received specification for its properties. Section 7 contains the discussion and conclusions.

2 Problem statement

In this section the context of the paper is introduced in detail. The role of specifications for standards is discussed here as well as the specific challenge to set up standards in the spatial domain. Then the previous work is presented and related to the contribution of this paper.

2.1 Specifications in the context of standards

The problem of building and maintaining large, i.e. complex and heterogeneous information systems — not only spatial ones — is a problem generally acknowledged by the software industry (Gibbs, 1994). The emerging trend to componentialization even increases the demand for clean software engineering methodologies, as different components will have to co-operate with each other.

Software engineering provides many techniques and tools supporting the software development process. One of the most important phases of this process is the specification (Liskov and Zilles, 1978; Liskov and Guttag, 1986). Specifications rely on abstraction, decomposition, and formal notation of a *conceptual model* and represent a model at the *logical* level of software design (Worboys, 1995). Specifications describe the *what* of pieces of a task ('what are the actors?', 'what are their relations?', 'what kind of actions have to be taken?'), not the *how* ('how do the actions have to be executed?'). Hereby, specifications do not only name the actions but also describe the restrictions, result and meaning of these actions, i.e. the semantic aspects of an action.

Specifications are essential for standards: standards typically have to describe *what* things are, how they should behave, what correct behavior is, but not how things should be implemented (data structures and algorithms). Such specifications do not make any assumption on technologies for implementation (*abstract specifications*).

In general, three properties of an abstract specification have to be verified:

1. It has to match with the problem to be specified.

   This property is made sure by agreement of experts (validation); thus the specification has to be communicatable to domain experts.

2. It has to be consistent in itself.

   This property can be checked automatically if and only if the specification is made in a formal language, and the degree of the check depends on the (type-) strictness of the language.

3. It has to be complete.

   This property can be checked semi-automatically via extensive testing, especially that of exceptions and strange cases. Testing can be alleviated if the specification is made in an executable language.

The latter two require a tool interpreting and executing the specification. Specifications in functional languages are executable; those in UML are not. However, also the validation of a functional specification (and of implementations) lacks for formal tools.

Standards are specifications for the whole software vendor and application development community. Their development exceeds some of the demands for a single software product specification by far:

- specifying experts and implementation experts can no longer communicate directly;
- full understanding of the specification has to be derivable from the specification documentation directly;
- specifications with a status of a standard should be consistent and error-free to avoid costly changes to product;
- specifications with a status of a standard need to have some durability (flexibility is restricted);

- specifications with a status of a standard need general acceptance (only in case of legal norms — like ISO standards — implementation can be prescribed).

The main challenge for standard specifications lies in unambiguity. Each vendor implementing a specification realizes an *interpretation* of the specification. For later interoperability of software, this interpretation has not only to be valid, it has to match in the semantics.

2.2 Complexity of standards for geographic information

Geographic information deals with physical, observable reality as well as social reality (Searle, 1995). The objects to be dealt with are partly bounded by physical boundaries (*bona fide*), partly bounded by agreement of people (*fiat*) (Smith and Varzi, 1997). Some of the *fiat* objects may have an exact definition, some have not. Definitions of objects (i.e. their boundaries) are a function of observation context (scale). Observation can never be precise. Thus, dealing with indeterminate boundaries, a variety of contexts and their ontologies is a constituent for geographic information (Burrough and Frank, 1996). Standardized specifications of spatial phenomena and services are essential for communicating geographic information (Frank and Kuhn, 1999; Kuhn, 1997).

In the field of geographic information, several institutions work on standards to achieve interoperability between different service components, among them OGC (Kottman, 1999), ISO (ISO, 2000), and several others. Also, there is work done to align different standards, e.g. between OGC and ISO (Østensen, 2000).

Interoperability standards for spatial information systems deal not only with data exchange formats between different systems, but define spatial entities as objects with behavior and include a services model such as for standard coordinate system conversion. Furthermore, standards integrate different forms of representation seamlessly. Thus, standards for geographic information systems model highly complex spatial worlds. Specification tools and techniques that support unambiguity, automatic consistency checking, and modularization are more necessary then ever.

2.3 Previous work

It becomes clear that the tools used for the specification need a capability to represent and to deal with semantic, since all readers of a specification should have an unambiguous understanding of the semantics of a specification. Current specification techniques in standards organizations are based on visual object oriented modelling in UML (OMG, 2000; Booch et al., 1999). UML neither has a formal semantics (see Bruel et al., 2000) nor allows formal modeling of the semantic aspects of expressions. The latter is usually done by comments in text form. For that reasons, models in UML are not executable.

The requirements in the field of geographic information recently lead to proposals to use functional languages for writing *algebraic specifications* (Egenhofer and Frank, 1992; Frank and Kuhn, 1995, 1999; Kuhn, 1997; Frank, 1999). Algebraic specifications (Guttag and Horning, 1978; Horebeek and Lewi, 1989; Loeckx et al., 1996) provide the following features:

- they have a mathematical clean form,

- they are sufficiently abstract,
- they are constructive, which means they are executable and their behavior can be checked, and
- modules can be combined to complex systems.

A recent investigation of many-sorted algebraic data models for GIS showed the usefulness of the approach, but stated the lack of a suited language (Lin, 1998). The critique on availability of suited tools is quite old (Backus, 1978), however, from our point of view functional programming has overcome the usability problems. Functional programming languages have a formally defined semantics (Stoy, 1977; Peyton Jones, 1987). Functional languages like Haskell are declarative, operational, and object-oriented (Peyton Jones et al., 1999; Hudak et al., 1999; Bird, 1998), and thus, fit the need for a usable tool to implement specifications as many-sorted algebras. The work of Kuhn and Frank demonstrates the usability of Haskell for capturing the semantics of expressions (Kuhn, 1994, 1997; Frank and Kuhn, 1999). Their interest in expressiveness is acknowledged here; however, a proof of practical applicability still has to be supplied.

2.4 Contribution of this work

We build on the work above, and use many-sorted algebras as basis for the specification of a spatial standard. Many-sorted algebras allow to specify a spatial standard in separated modules of, for example, features, geometry, spatial reference systems, etc., and compose them to a whole. Thereby, we use Haskell as a specification language since Haskell supports the definition of operations as pure mathematical functions, and also allows to capture the semantics of these operations formally. Furthermore, it allows to compile, check and execute the specification.

Our concern in this work is to use the approach above, and investigate its practicality and usefulness by selecting a real-world spatial standard, and specifying the syntactic and semantic aspects of a complete section of this standard formally via Haskell. We look at questions like: If this approach is advantageous for the above reasons, does it create at the same time new problems? Is the standard development process different then? What are the consequences for the creation, the maintenance, and the extension of a standard? Are the efforts for a formal standard competitive in time and costs? Does the approach have any impact on the domain? We will report on our findings and compare them with the practice in standards development.

3 Standards by OGC: The OpenGIS coverage specification

In this section, we shortly describe the process of how OGC defines specifications. Following, we will introduce the grid coverage as described in the essential model to introduce the terminology for the classes of the domain. We also give a short example of the OGC's UML specification and the resulting implementation specification, and discuss this approach.

3.1 The specification process of OGC

OGC is a consortium of industry, academic and government members who meet regularly to cooperatively define the OpenGIS Model (Kottman, 1999), a standard for interoperability of GI services and data across different platforms.

In a first phase, the OGC members defined the vision, scope and content of the model (i.e. a seamless model for vector geometry and rasters as well as a service model). The result is documented in the *abstract specification*. The abstract specification does — like a specification during the software engineering process — describe the problem and tasks without making any assumptions of possible technologies that might be used to implement it. The abstract specification consists of two parts: a verbal description, the *essential model*, and a semi-formal *abstract model* using UML. Today, the abstract specification consists of 16 chapters.

To achieve practical interoperability, GIS software itself has to be interoperable; thus, based on the abstract specification, OGC uses a Request-for-Proposal-process to call for *implementation specification* proposals from its members[1]. An implementation specification uses the abstract specification, and makes a suggestion of how its concepts should be best implemented using most likely technologies such as SQL, Java, XML, Microsoft's COM/OLEDB, or OMG's CORBA. The selected technologies depend on the state of art of widely-used distributed computing platforms. These might change over time, and new implementation specifications might be added still based on the existing abstract specification. There is one information specification per platform, and the proposers have to make sure that the implementation specifications for different platforms are consistent, especially semantically consistent. Implementation specification proposals are reviewed by the OGC members and voted on for acceptance as standard. Implementation specifications are maintained and aligned with each other during a revision process. At this point, companies start building products based on an implementation specification that exhibit the implementation specification interfaces. Products can then be branded OGC-compliant if they fulfill an OGC conformance test.

For this paper, we select the coverage chapter of the abstract specification (OGC, 2000b), and concentrate on grid coverage, for which recently an implementation specification has been adopted (OGC, 2000a).

3.2 The OpenGIS essential model for coverage

Coverage is coined by OGC for all kinds of field representations: grid, image, TIN, polyhedral surface, and more. Coverage is defined as a special subtype of a *feature*. In the OpenGIS model feature is the main concept for any kind of spatial entity. A feature has properties defined as a list of pairs of *property name* and *property value*.

A coverage contains a special property, a coverage function (*cfunction*). (A coverage might have one or more cfunctions.) A cfunction is a function that has a spatial domain, and a set of values as range. Typically, the spatial domain of the cfunction is a set of (any) geometries. (In the OpenGIS model,

[1]For practicality reasons, these RFPs are for well-defined subparts of the abstract specifications. For example, the implementation specification for geometry was limited in a first step to the so-called simple features.

geometries are (sets of) any kind of point, line, or polygon.) For each geometry in the spatial domain, the cfunction evaluates the *value* at this location. The general concept of a value is a vector v of single values of different types. An example would be the temperature and the rainfall at a given point. Thus, the range of a cfunction is a set of homogeneous vectors. When a cfunction has a continuous spatial domain, it calculates a value for each point in the spatial domain via interpolation. In a simpler case, if the spatial domain is discrete, i.e. it encompasses a finite set of geometries, the cfunction is called a *discrete cfunction*. Finally, if the spatial domain is restricted to a finite set of *points*, we talk about a *discrete point cfunction*.

Grids are closely related to discrete point cfunctions. In the special case of a grid geometry, the domain of the cfunction is a finite set of points in regular order. Using the grid geometry as a spatial domain, the abstract specification introduces another specialized discrete point cfunction, the *grid value matrix*. The grid value matrix consists of a set of *point-value-pairs* whereby the points form a grid. A grid coverage is a specialized coverage containing a grid value matrix as specialized cfunction. Furthermore, a grid coverage has an *evaluator* which evaluates the value for each grid point. If the domain of a grid coverage is extended to the convex hull of a grid, the evaluator can implement an interpolation function to calculate values for each point in the convex hull. Several interpolation algorithms are used today whereby nearest neighbor is the most common, and it is also the default function in the OpenGIS specification.

3.3 The OpenGIS abstract model for coverage

The essential model above verbally captures the scope of our example. In a second step, the objects of the essential model, their relations, and their properties are abstracted and represented in UML. We select from OGC's abstract model the cfunction and the discrete cfunction to demonstrate the properties of such a semi-formal representation.

Figure 1 depicts a UML representation of the `CFunction`, and its relationship to `Coverage`. In UML, the arrow between `Coverage` and `CFunction` represents a dependency relationship between an independent `Coverage` and a dependent `CFunction`. However, this presentation is insofar ambiguous since the relation is not named; if it should represent an association it does not contain multiplicity information: has a coverage exactly one, or at least one `CFunction`? Furthermore, the diagram shows that a `CFunction` has two methods, `evaluate` and `domain`, and specifies their input and output data types. Invisible here but attached to the diagram a textual description of parameters and methods can exist. These textual descriptions are repeated in the textual part of the abstract model making them visible in the document. The semantic aspect is still not captured formally.

In Figure 2 is modeled that a `DiscreteCFunction` is a special case of a `CFunction`; thus, `DiscreteCFunction` inherits all methods from `CFunction` and is able to overwrite them, and new methods can be added. (In UML, the semantics of an inheritance arrow is clearer than that of a dependency.)

Figure 3 depicts a `DiscreteCFunction` in a context with `Coverage`. One can see that the domain of a `DiscreteCFunction` is discrete and finite, since it returns a finite list of geometries. However, the presentation does not capture which geometry arguments are acceptable for the `evaluate` function. It might

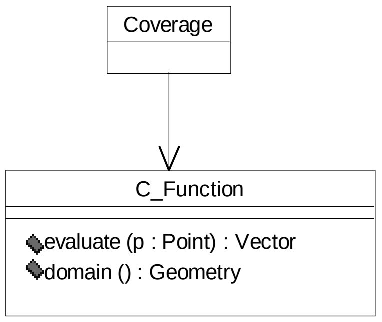

Figure 1: The UML context diagram of OGC for `CFunction`.

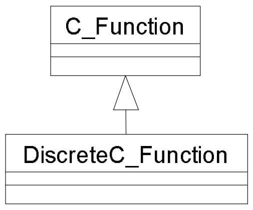

Figure 2: The UML class diagram of OGC for `DiscreteCFunction`.

be that any geometry types are acceptable, or only geometry objects of the type found in the function's domain.

3.4 Discussion of the OGC approach

This process demonstrates that the use of UML in the abstract specification is only semi-formal with important portions of the specification written in English. The UML part is open to interpretations, and so far ambiguous. Furthermore, the correctness of the UML model cannot be guaranteed or tested.

A quick comparison of the draft implementation specification for grid coverage with the presented abstract model shows changes in the implementation. The non-formal derivation of the implementation specification from the abstract specification allows implementors to interpret the abstract specification, and make decisions that are not aligned with the intention of the abstract model. Where the implementation specification can solve problems with an imperfect abstract specification, a revision of the abstract specification should be initiated. Additionally, implementation specifications for different platforms have to be aligned, which is also a process managed informally in OGC. Much of the problems would be solved if implementation specifications were derived from the abstract model automatically for all platforms (Berre et al., 2000). Again, this requires a formal abstract model and a way to check the correctness and completeness of the abstract model. For these reasons we propose another, more formal tool for the abstract specification.

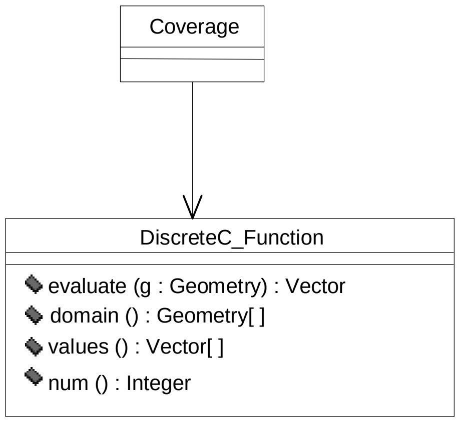

Figure 3: The UML context diagram of OGC for `DiscreteCFunction`.

4 Functional languages for spatial standards

Here, we recapitulate algebraic specification as one of today's advanced software engineering techniques. We choose a functional programming language to write our algebraic specifications. Thus we will present the principles of the language also.

4.1 Algebraic specifications

An algebra captures the coordinated behavior of a set of operations that are applied to the same object. An algebra consists of three parts: a type, a set of operations, and the behavior (or semantic) of the operations which is defined by axioms. An algebra does not describe what the objects are but how they behave, and captures the semantics of this behavior. Algebras can be used to specify abstract data types by behavior, and do not determine the implementation (Horebeek and Lewi, 1989; Loeckx et al., 1996). Consider for example a possible algebra of *points*, given by a type (or *sort* in terms of algebra) `Points`, two operations `getX`, `getY` on the type, which return the coordinates of a point, and an axiom for equality of two points a and b: `equal a b = (getX a == getX b) && (getY a == getY b)`. Note that the axiom is independent from the internal implementation of points; it defines behavior (or semantics) only.

Many-sorted algebras (Guttag and Horning, 1978; Horebeek and Lewi, 1989; Loeckx et al., 1996) are structures consisting of some sets of objects (types) and a number of functions whose arguments and results belong to these sets (Horebeek and Lewi, 1989, p. 15). Consider a second type `Lines`. If there are operations `start` and `end` to return the start and end point of a line, then an axiom `concatenated` can be defined that checks whether two lines are concatenating. The type declaration would be `concatenated :: Lines -> Lines -> Bool`, and the specification of the behavior could look like `concatenated a b = (start a == start b) || (start a == end b) || (end a == start b) || (end a == end b)`.

Consider an example related to OGC specifications, on (spatial) *features* and their *properties*. For OGC a feature has an identifier and a list of properties; one of the properties is the geometry of the feature, which can be accessed directly. The algebra of `Features` utilizes objects of the type `Properties`; it provides operations and axioms on features. Typical operations could be `getFeatureProperties: Feature -> [ Property ]`, where the square brackets denote a list, or `getGeometry :: Feature -> Property`. A second algebra specifies operations and axioms for `Properties`. Then both algebras can be composed with no internal adaptation.

Composability of many-sorted algebras allow a piece-wise development of the specification. The pieces can be composed as long as types are pair-wise disjoint and no naming conflict occurs. Composability is essential for specifications in complex domains like the spatial. It is part of the abstraction of the domain (Liskov and Guttag, 1986). In executable algebraic specifications pieces can be tested separatly for correctness and completeness, and afterwards combined to more complex parts.

4.2 Functional programming languages for algebraic specifications

Using functional programming languages (Backus, 1978; Henson, 1987; Jones, 1995; Frank and Kuhn, 1995; Bird, 1998) for writing specifications, the mathematical foundation of many-sorted algebras is preserved, and additional advantages are obtained from automatic processing and interpreting the code. Functional languages appear as natural means to describe algebras. Without further emphasis, the examples above were already given in the syntax of Haskell, the functional language of our choice. The code for, e.g., points looks like:

```
class Points p where
  x     :: p -> Float                         -- operation declaration
  y     :: p -> Float                         -- operation declaration
  equal :: p -> p -> Bool                     -- axiom declaration
  equal a b = (x a == x b) && (y a == y b)    -- axiom definition
```

An algebraic specification written in Haskell consists of classes (for the types of the algebras) and functions that describe *axioms* as well as *operations*. Functions defined at class level represent axioms, and functions defined at instance level represent operations. Consider the example of points. The function `equal` could be defined on class level. For the operations returning coordinates we need an instance for an explicit data type:

```
data Point = Pt String Float Float
instance Points Point where
  x (Pt name xval yval) = xval                -- operation definition
  y (Pt name xval yval) = yval                -- operation definition
```

Compiler or interpreter for the language first parse the code. Code accepted by a parser is written in correct syntax. Further checks depend on the specific properties of the chosen language. We decided to specify in Haskell (Hudak et al., 1999; Bird, 1998), or, to be precise, in an implementation of Haskell called Hugs (Jones et al., 2000). Haskell has a number of properties recommending its use for algebraic specifications:

- Haskell is a strongly typed language, i.e., each type has to be unambiguous at compile time. For that reason, code accepted by a compiler or an interpreter guarantees type consistency additionally.

- Related to the type concept, Haskell allows no side effects, which eliminates unforseen interactions and makes the behavior transparent.

- Haskell is a polymorphic typed language. Classes and functions can be instantiated for different types, which increases re-usability of code. Moreover, the class concept of Haskell supports multiple inheritance and overloading; this will be exploited in the following for modularization.

- Modularity accomplishes the requirement for abstraction and decomposition. Furthermore, it resolves the naming problems combining different algebras by providing qualified namespaces.

- Haskell is a declarative language. It provides the necessary level of abstractness for specifications by describing exactly the *what* and not the *how*. This makes the code extremely compact and readable, compared to imperative languages. Declarativeness captures the semantics of function signatures.

- The class concept and declarativeness allow to specify constructors that extend an initial specification, consisting of a class name and the declaration of its functions only, to a constructive specification (Loeckx, 1987; Loeckx et al., 1996). Constructive specifications are executable; i.e., constructive specifications represent at the same time prototypes of the specified program. Constructive specifications have an operational semantics which is the behavior of the program.

Consider points again. An executable specification can be tested, like:

```
a, b :: Point
a = Pt "pointA" 45.9 57.6
b = Pt "pointB" 49.3 53.1
test = equal a b
```

In the next section we use Haskell for a many-sorted algebraic constructive specification of the grid coverage.

5 A functional version of the OpenGIS coverage specification

In this section the coverage abstract model is presented in an alternative form, as an application of Haskell. In principle, there are two approaches to achieve a formal specification from the OGC documents:

1. One can start from the essential model, then extract the objects and the relations from the essential model, and put them into a functional language.

2. One can start from the abstract model, taking the objects and relations and translate them into a functional language.

The first approach yields a specification which is nearly independent from the abstract model of OGC. Differences would relate more to different abstractions and conceptualizations than to different capabilities of the used tools UML and Haskell. The second approach relies on the abstraction in OGC's abstract model. Thus it has to cope with the ambiguity and inconsistency of the UML specification. However, the goal of this investigation is a comparison of expressiveness and practicability of tools and not of the specifications, and therefore the second approach is chosen. The transition from UML to the functional language is described by examples that show the similarities and the differences compared to the UML model. These similarities and differences are discussed.

5.1 The classes and function declarations

In this section the UML classes and operations are translated into Haskell classes and function declarations. Behavior (function definitions) will be added in the next section.

Consider the (UML-)class `CFunction` in Figure 1. Representing a class `C-Functions` in Haskell looks like:

```
class CFunctions cf where
   evaluate :: cf -> Geometry -> Vector
   domain   :: cf -> [Geometry]
```

The class contains the declaration of two functions, `evaluate` and `domain`. The function `evaluate` takes a `CFunction` of the abstract data type `cf` and a geometry, and returns a vector of function values. The implicit meaning that the vector contains the values *at the given location of the geometry* will be explicated later. The function `domain` takes a `CFunction`, and returns a list of geometries. Again the meaning that the geometries represent the domain of the coverage function is explicated later.

Declarations introduce the names of functions and their parameter types. So far the code example shown above has exactly the same power of expressiveness as the UML code of Fig. 1. A minor difference (which is not visible in this example) is evident in the constraint that in Haskell everything has to be written as a function: even parameters in UML classes will be functions in Haskell (the UML class `CFunction` contains no parameters). From the perspective of an *abstract* specification — i.e., a specification with no reference to implementation specific details or architectures — this is an advantage: the abstract specification in Haskell is restricted clearly to interfaces, it does not allow decisions how to code parameters.

A closer look at the example shows significant differences between the UML and the Haskell specification of a `CFunction`:

- We decided to describe the domain via a *list* of geometries, where the UML specification shows only a single geometry. This is not a contradiction because OGC considers lists of geometries to be a geometry (GeometryCollection) again. However, in the class `DiscreteCFunction` (Fig. 3) the UML code uses lists for the domain, too. We need a consistent notation for the type checking system of Haskell.

- Another difference can be found in the parameter type of the `evaluation` function in `CFunctions`. OGC's UML code shows a point where we decided to put in a geometry. One reason is again type consistency with the inheriting class `DiscreteCFunction`. Another reason is an inconsistency in OGC's text with its UML code; the text demands that the function parameter has to use the geometry type of the domain of the `CFunction`, otherwise the function should return an error.

The correct specification of the UML-class `DiscreteCFunction` (Fig. 3) is in Haskell:

```
class CFunctions a => DiscreteCFunctions a where
   num    :: a -> Int
   values :: a -> [Vector]
```

The functions `evaluate` and `domain` are inherited and need not to be declared again.

Following the shown approach the UML model of OGC can be translated into Haskell with the same power of expression but more consistent.

5.2 Semantics

Semantics of declarations in UML as well as in Haskell are captured mostly in the names of the declared classes and functions. The abstract model of OGC uses extensive verbal descriptions to document the meanings of classes and the behavior of their interfaces. These descriptions need to be interpreted by readers of the abstract model. In contrast, Haskell allows to specify semantics in a formal manner.

Given a declaration of the function `domain` (see above), one can add its behavior by:

```
   domain cf = map geom (getGVPs cf)
```

`getGVPs` is an operation in the algebra of cfunctions returning the geometry-value-pairs that form the cfunction. The function `geom` is an operation in the algebra of geometry-value-pairs. It returns the geometry for a given geometry-value-pair. Finally, `map` is an operation in the algebra of lists, which is in-build in Haskell. It maps a function on each element of a list, returning the list of function results. Thus the code defines the semantic of `domain` by stating that the domain of a cfunction is the list of all geometries that are contained in the cfunction's defining geometry-value-pairs.

Similarly the two functions of `DiscreteCFunctions` can be defined, specifying their semantics formally:

```
   num dcf    = length (domain dcf)
   values dcf = map value (getGVPs dcf)
```

The `num` method returns the cardinality of the domain. This is determined by the length of the list of all geometries, which is finite since `DiscreteCFunction` has a finite domain. The `values` method returns the list of values of all geometry-value-pairs that define the `DiscreteCFunction`.

The examples show that implementations in Haskell specify behavior, not algorithms. By that, function implementations remain compact and comprehensive, i.e. the formal specification of semantics can be communicated.

5.3 Abstraction

The Haskell examples from above were kept simple for comprehensability. A functional programmer would probably start with them, but then the code can be improved with the goal to separate an *abstract specification* from all implementation issues. The above examples still contain references to concrete data structures in the typed declarations, namely `Geometry` and `Vector`. The use of data types can be eliminated by introducing abstract classes (`Geometries`, `Vectors`) which describe the behavior of abstract types (`g`, `v`). After revision the specification is purely abstract:

```
class (Geometries g, Vectors v)
      => CFunctions a g v | a -> g v
      where
   evaluate        :: a -> g -> v
   domain          :: a -> [ g ]
```

Thus, this specification is about semantics of types, not about their implementation, a desirable property for abstract specifications. Implementation decisions for data types can be shifted completely to the realization of classes in instances, for example:

```
instance CFunctions CFunction Geometry Vector
```

where the data types `CFunction`, `Geometry`, and `Vector` replace the abstract data types of the specification. Abstraction in this way shifts commitments for data types through dependency chains to base classes. Base classes have no further dependencies. In the coverage specificaiton, an example of a base class is `PropertyValues`. This class is needed only for fixing a data type, and no functions need to be specified:

```
class PropertyValues a
instance PropertyValues PropertyValue
```

Thus the *abstract specification* can be limited to *class declarations* only. This set of class declarations can be checked for type consistency. However, pure class declarations are not executable. Adding *instances* completes the semantic specification and links to data types for executability. Instances are comparable to *implementation* specifications of OGC, if they are considered as interpretations of the abstract specification for a programming environment. Here the environment is Haskell, and we remain still independent from specific technologies like CORBA or COM. Abstract and implementation specification together can be executed and tested.

5.4 The grid coverage in Haskell

Following the approach presented above the grid coverage section from OGC's specification *Coverage and its Subtypes* (OGC, 2000b) was transferred into Haskell. In this section, we present our findings.

The complete code of the formal specification is available via ftp from `ftp://-ftp.geoinfo.tuwien.ac.at/winter/`. The unzipped package can be viewed by a normal text editor, and it can be tested using Hugs (Jones et al., 2000), which is public domain software.

Coverage and grid coverage are specified in about 20 modules/classes. The size of the pure abstract specification is compacter than OGC's abstract model. However, we added constructors and test data for test cases, which increases the volume to about 20 A4 pages.

Characteristics of the Haskell specification are:

- A class hierarchy `CFunctions`, `DiscreteCFunctions`, `DiscretePointC-Functions`, and `GridValueMatrices`. `GridValueMatrices` are composed by array collections. Each array has to have a constant data type, and the arrays have to have the same origin, orientation, resolution and size. Orthogonal to the matrices, `PointValuePairs` exist which represent the values at one grid position (or, if the evaluation function allows interpolation, at any point in the grid coverage area).
- A class hierarchy `Feature`, `Coverage`, and `GridCoverage`. The `GridCo-verage` is a feature represented by a (distinctive) property with the name `"CFunction"` and the value `GridValueMatrix`.
- A class for `AnyType`, a complex data type that can be anything, a number, a string, a geometry, or a `CFunction`. This type is used for the property values of feature properties. Methods of the class `AnyType` allow to determine the actual type as well as the value of the property.

One goal of the formal specification in Haskell is the proof of practicality. For that reason, we cover only a well-defined subset of the abstract specification, namely coverage and grid coverage, and set out to achieve similar completeness for the other parts of this abstract model.

6 Properties of the functional specification

The Haskell package representing an abstract model of coverage and grid coverage is executable and can be tested. We report here about the proof of consistency and about the checks for correctness and completeness.

6.1 Consistency

Having the specification in a formal typed language, (internal) consistency can be checked automatically. The Haskell interpreter checks syntax and grammar, not the semantics. Take again the `domain` example:

```
domain :: cf -> [Geometry]
domain cf = map geom (getGVPs cf)
```

While loading the module, the Haskell interpreter parses the code (it is valid Haskell, so no objections will occur) and checks the chain of types in the called functions. To do so the function declarations are utilized:

→ The method `getGVPS` takes a `CFunction` and returns a list of `Geometry-ValuePairs`.

→ The method `map` takes a function (here: `geom`) and maps it onto a list. The result is a list of the output type of the function: `geom` takes a `GeometryValuePair` from the list and returns a `Geometry`.

→ That means the method `domain` takes a `CFunction` and returns a list of `Geometries`.

The consistency check for `domain` will succeed, because the types fit together, and the chain matches with the declaration of `domain`. Because the check of consistency passes automatically and for the complete code, we can state that consistency is proven if a module or package is loaded successfully. — Contexts and type dependencies in abstracted specifications load a lot of additional type checking work to the interpreter, but the tasks are mainly the same, and type conflicts will be recognized by the interpreter.

Specification tools for UML support checking of consistency not with this strictness. Neither the language UML is a typed language, nor an interpreter for UML is available.

6.2 Correctness

Some of our interpretations of the essential model of OGC may give reason for discussion. This is a desirable property of a formal specification: it should improve communication between domain experts (i.e, the authors of the essential model) and formalists (the authors of the abstract model). If contradictions between the two models or misinterpretations of the essential model can be easily identified and eliminated, the specification method is useful. Furthermore, the abstract model in Haskell is executable. It allows testing of behavior, i.e. domain experts only have to check and to accept the behavior. Behavior is communicated and discussed in form of test cases (critical ground terms) which can be set up in Haskell easily. Execution of the tests check that the values of the test cases are the values expected.

Our Haskell specification shows some differences to OGC's abstract model in UML. This is due to the fact that we were required to make design decisions that captured not only the parts but also the relationships between the parts in a coherent, and compilable way. Additionally we specified constructors among the operations, creating constructive specifications. In this sense, we produced a specification that is closer to an implementation specification, and thus, narrow the gap between an abstract and an implementation specification.

The development of the Haskell specification showed significant shortcomings of OGC's abstract model. These shortcomings exist regarding definitions that could be consistent, but are not correct when tested. Examples are:

- Evaluation of coverage. All types of `CFunctions` use a multi-type value vector. However, the `Evaluator` represents one single evaluation (or interpolation) method, which is not useful for multi-type value vectors (multi-type vectors are allowed explicitly).

- Inheritance of properties, like `Evaluator` or `SpatialReferenceSystem`, through the hierarchy of `CFunctions`, `GridValueMatrix`, and `GridCoverage`, is not evident in the abstract model. That leaves no significant difference between `GridValueMatrix` and `GridCoverage`, because all required properties for the `GridCoverage` are taken from higher classes and are not unique for the `GridCoverage`.

What cannot be checked in the consistency check is the behavior of functions. This exceeds internal correctness and refers to a correct formal representation of another model, the essential model of the domain. Only the domain expert can evaluate the behavior; no automatic routine can guarantee correctness.

One possibility to check behavior is reading, understanding and accepting the code. Compared to imperative programming languages, declarative code is extremely compact and could be understood easily by Haskell programming experts. However, because the validation of behavior has to be done in coordination with domain experts, code may not be the best way to communicate. An alternative is to check behavior by examples. Choosing typical, simple or even trivial examples from the specified domain eases communication with domain experts in evaluating the result. Consider again the `domain` method. Given any simple `CFunction` the method `domain` can be applied on it, and the result can be evaluated:

```
cf :: CFunction
cf = ...
test :: [Geometry]
test = domain cf
```

We conclude that tests of behavior are easy to provide in an interpreter environment. Our Haskell modules typically have implemented some test functions at the end of each file; they can be called directly.

6.3 Completeness

We made several short cuts in the functional specification. For instance, `Features` are simplified here, and also we do not consider non-regular `DiscretePointCFunctions`. Doing so we achieve a (more or less) complete functional specification with regard to the chosen topics from the essential model. Additions can be made by merging other algebras, e.g., for `Features`. Merging algebraic specifications do not require any changes inside of given specifications. This is especially true for specification tools that provide namespaces, like Haskell with its concept of qualified import.

After the agreement that the specification reflects the purposes of the essential model, tests can be continued to check the treatment of extremal test cases. The procedure is in principle the same as before. As a result, it could be stated which extremal cases or exceptions are covered by the specification.

7 Discussion and conclusions

7.1 Summary

We proposed a formal tool for specifying spatial standards, using a functional programming language for writing many-sorted algebras. We exemplified the applicability for a topic from the OpenGIS Consortium's bookshelf. Our formal version of that abstract model could be compared in detail and as a whole with OGC's abstract model as well as with OGC's implementation model for the grid coverage. We verified some (expected) advantages of the formal specification: a formal semantics of specified interfaces, an executable prototype, guaranteed consistency, an extendable algebraic structure, a total abstraction from implementation. Additionally, we exemplified the complexity of abstract formal code, and we argued that communication of the specification should be made through its behavior instead of its code.

Our formal specification remains incomplete with regard to OGC's coverage specification. With the presented specification procedure, other parts of OGC's specification could be added, like triangulated irregular networks, or Thiessen polygons. We remain incomplete also with regard to cuts at the limits of our selected topic; for example, the class `Features` is realized only in its most primitive version. `Features` are treated in another book of OGC. A many-sorted algebra for `Features` can be added easily, replacing our poor version. Hugs supports the composition by separating the namespaces of modules.

The provided formal specification contains constructor functions and utilizes them for test cases. Test cases show the behavior of the specification. Their real value lies in the possibility for implementers to check the behavior of their implementation against the behavior of the specification. Conformance testing — up to now an external and expensive process — could make profit from this property by delivering the conformance test cases together with the specification for in-house testing.

7.2 Conclusions and future work

We have shown that formal specifications in functional languages are manageable and turn out to be valuable. The applied specification technique is superior to semi-formal visual modeling in UML due to formal semantic, consistency, and executability. It is to be expected that such specifications are cheaper in application development and conformance testing. Even more important, it is to be expected that such specifications are easier to maintain, due to automatic checks and direct testing capabilities. Therefore, it can be recommended to apply the technique in spatial standards organizations.

From a scientific perspective, we have shown how to separate abstract and implementation parts in formal specifications. Further we have made clear the differences between a semi-formal and a formal specification, and developing the formal version forced us to solve the inconsistencies in the existing semi-formal specification. We see the following open questions.

- Can abstract modeling in Haskell be supported by a visual tool similar to UML?

  Looking at our procedure — we took the UML classes and methods as they are —, we expect that a visual tool similar to UML-tools would help to create specifications from scratch. We expect also that such an improvement of the user interface would improve the acceptance of Haskell in the standards community.

- Can a formal specification be transformed automatically in less formal specifications, e.g. in UML?

  This idea could improve the acceptance, too. In principle, there is no problem, because class names, class dependencies, and function names and types can be taken directly from the functional code. However, putting this in UML provides a less formal specification, with the loss of all behavioral information. A special problem is the translation of formal function definitions into readable texts.

- Can an OGC implementation specification be derived automatically from Haskell?

  Implementation specifications are provided for several technologies by OGC. It would be desirable to derive these specifications automatically, i.e. aligned in their semantic.

- How far can we go with Haskell?

  We have shown that a single topic of OGC's bookshelf can be specified in Haskell. Also we have stated that other algebras can be added without any difficulty. Further, in Haskell a specification is identical to its prototype. If once the complete bookshelf is specified in Haskell, one has at the same time a GIS prototype conforming with the OGC standard (better: realizing the OGC standard).

Acknowledgements

Gerhard Navratil and Andrew Frank, both TU Vienna, gave valuable input and ideas to improve the functional specification. Also we gratefully acknowledge the help of Cliff Kottman, OGC, in interpreting OGC's abstract specification Topic 6: The Coverage and its Subtypes.

References

Backus, J., 1978. Can programming be liberated from the von neumann style? A functional style and its algebra of programs. Communications of the ACM 21, 613–641.

Berre, A. J., Skogan, D., Gronmo, R., Solheim, I., Hoff, H., Skjellaug, B., 2000. Practising the ISO/TC 211 approach to GI interoperability. In: Space Applications Institute (Ed.), 6th EC-GI&GIS Workshop Lyon. European Commission, Joint Research Centre, Brussels.

Bird, R., 1998. Introduction to Functional Programming Using Haskell. Series in Computer Science. Prentice Hall Europe, Hemel Hempstead, UK.

Booch, G., Rumbaugh, J., Jacobson, I., 1999. The Unified Modeling Language Reference Manual. Addison-Wesley, Reading.

Bruel, J.-M., Evans, A., France, R., Lano, K., Kent, S., Moreira, A., Rumpe, B., 2000. The precise UML group home page. Tech. rep., http://www.cs.york.ac.uk/puml/index.html.

Burrough, P. A., Frank, A. U. (Eds.), 1996. Geographic Objects with Indeterminate Boundaries. Vol. 2 of ESF-GISDATA. Taylor & Francis.

Egenhofer, M. J., Frank, A. U., 1992. Object oriented modeling for GIS. Journal of the Urban and Regional Information Systems URISA 4 (2), 3–19.

Frank, A. U., 1999. One step up the abstraction ladder: Combining algebras — From functional pieces to a whole. In: Freksa, C., Mark, D. M. (Eds.), Spatial Information Theory. Vol. 1661 of Lecture Notes in Computer Science. Springer-Verlag, Berlin, pp. 95–107.

Frank, A. U., Kuhn, W., 1995. Specifying open GIS with functional languages. In: Egenhofer, M. J., Herring, J. R. (Eds.), Advances in Spatial Databases. Vol. 951 of Lecture Notes in Computer Science. Springer, Berlin, pp. 184–195.

Frank, A. U., Kuhn, W., 1999. A specification language for interoperable GIS. In: Goodchild, M. F., Egenhofer, M., Fegeas, R., Kottman, C. (Eds.), Interoperating Geographic Information Systems. Kluwer, Norwell, MA, pp. 123–132.

Gibbs, W. W., 1994. Software's chronic crisis. Scientific American 271 (3), 72–81.

Gruber, T. R., 1993. Toward principles for the design of ontologies used for knowledge sharing. Technical report ksl 93-04, Knowledge Systems Laboratory, Stanford University.

Guarino, N., 1998. Formal ontology and information systems. In: Guarino, N. (Ed.), 1st International Conference on Formal Ontology in Information Systems. IOS Press, Trento, Italy.

Guttag, J. V., Horning, J. J., 1978. The algebraic specification of abstract data types. Acta Informatica 10, 27–52.

Henson, M. C., 1987. Elements of Functional Languages. Computer Science Texts. Blackwell Scientific Publications, Oxford.

Horebeek, I. V., Lewi, J., 1989. Algebraic Specifications in Software Engineering. Springer-Verlag, Berlin.

Hudak, P., Peterson, J., Fasel, J. H., 1999. A gentle introduction to Haskell 98. Tech. rep., http://www.haskell.org/tutorial/.

ISO, 2000. ISO/TC 211 programme of work. Tech. rep., http://www.statkart.no/isotc211/pow.htm.

Jones, M. P., 1995. Functional programming with overloading and higher-order polymorphism. In: Jeuring, J., Meijer, E. (Eds.), Advanced Functional Programming. Vol. 925 of Lecture Notes in Computer Science. Springer, Berlin, pp. 97–136.

Jones, M. P., Reid, A., Yale Haskell Group, Oregon Graduate Institute of Science and Technology, 2000. Hugs online. The Hugs 98 system home page, http://www.haskell.org/hugs/.

Kobryn, C., 1999. UML 2001: A standardization odyssey. Communications of the ACM 42 (10), 29–37.

Kottman, C. A., 1999. The Open GIS Consortium and progress toward interoperability in GIS. In: Goodchild, M. F., Egenhofer, M., Fegeas, R., Kottman, C. (Eds.), Interoperating Geographic Information Systems. Kluwer, Norwell, MA, pp. 39–54.

Kuhn, W., 1994. Defining semantics for spatial data transfers. In: Waugh, T. C., Healey, R. G. (Eds.), Advances in GIS Research – Proc. 6th Int. Symp. on Spatial Data Handling. International Geographical Union, Edinburgh, pp. 973–987.

Kuhn, W., 1997. Approaching the issue of information loss in geographic data transfers. Geographical Systems 4 (3), 261–276.

Lin, F.-T., 1998. Many sorted algebraic data models for GIS. International Journal of Geographical Information Science 12 (8), 765–788.

Liskov, B., Guttag, J., 1986. Abstraction and Specification in Program Development. The MIT Electrical Engineering and Computer Science Series. MIT Press, Cambridge, MA.

Liskov, B., Zilles, S., 1978. An introduction to formal specifications of data abstractions. In: Yeh, R. T. (Ed.), Current Trends in Programming Methodology. Vol. 1. Prentice-Hall, Englewood Cliffs, N.J., pp. 1–33.

Loeckx, J., 1987. Algorithmic specifications: a constructive specification method for abstract data types. ACM Transactions on Programming Languages and Systems 9, 646–685.

Loeckx, J., Ehrich, H.-D., Wolf, M., 1996. Specification of Abstract Data Types. Wiley-Teubner, Chichester.

OGC, 2000a. OpenGIS grid coverages implementation specification. Implementation Specification Draft Version 1.0, Open GIS Consortium.

OGC, 2000b. The OpenGIS specification model, topic 6: The coverage type and its subtypes. Tech. Rep. 00-106, Open GIS Consortium, Inc.

OMG, 2000. UML resource page. Tech. rep., http://www.omg.org/uml/.

Østensen, O., 2000. Joint steering group on spatial standardization and related interoperability. Tech. rep., http://www.spatialstandards.org/.

Peyton Jones, S., Hughes, J., Augustsson, L., Barton, D., Boutel, B., Burton, W., Fasel, J., Hammond, K., Hinze, R., Hudak, P., Johnsson, T., Jones, M., Launchbury, J., Meijer, E., Peterson, J., Reid, A., Runciman, C., Wadler, P., 1999. Haskell 98: A non-strict, purely functional language. Tech. rep., http://www.haskell.org/onlinereport/.

Peyton Jones, S. L., 1987. The Implementation of Functional Programming Languages. Prentice Hall International Series in Computer Science. Prentice Hall, New York.

Searle, J. R., 1995. The Construction of Social Reality. The Free Press, New York.

Smith, B., Varzi, A. C., 1997. Fiat and bona fide boundaries: Towards an ontology of spatially extended objects. In: Hirtle, S. C., Frank, A. U. (Eds.), Spatial Information Theory (COSIT '97). Vol. 1329 of Lecture Notes in Computer Science. Springer, Laurel Highlands, PA, pp. 103–119.

Stoy, J. E., 1977. Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. MIT Press, Cambridge, Massachusetts.

Worboys, M. F., 1995. GIS: A Computing Perspective. Taylor & Francis, London.