# Supporting the Construction and Use of Spatio-Temporal Domains In Scientific Databases

Terrence R. Smith* Jianwen Su*† Michael F. Worboys

{smithtr,su}@cs.ucsb.edu
Department of Computer Science
University of California
Santa Barbara, CA 93106
USA

michael@cs.keele.ac.uk
Department of Computer Science
Keele University, Keele
Staffordshire, ST5, 5BG
U.K.

July 11, 1993

*Work supported in part by NSF grant IRI-9117094.
†Work also supported in part by NSF grant IRI-9109520.

**Abstract**

A modeling and database system (MDBS) provides computational support that facilitates the iterative development of scientific models in data-intensive and computationally intensive applications in various areas of scientific application. We describe and exemplify core elements of an MDBS that is based both upon a simple characterization of scientific activity and upon the results of joint research with a group of EOS earth scientists. The MDBS that we describe is focused in particular on the representation and manipulation of large numbers of complex, spatio-temporal entities. The main contributions of the current paper are an advanced and comprehensive data model and a high-level database and modeling language (MDBL). The data model is based on a simple characterization of scientific activity that captures the essence of scientific modeling and database activities in terms of four fundamental sets of activities. These activities are defined in relation to large lattices of *representational domains* (R-domains) that are employed to organize scientific knowledge, data and algorithms. The data model separates the specification and representation of R-domains of elements, and allows multiple representations that are explicitly controlled by the user. Datasets and transformations are organized in terms of an extensible lattice of abstract and concrete R-domains. In particular, the data model allows the easy definition and manipulation of large numbers of domains of spatio-temporal elements. We present many examples of such domains from our applications. MDBL is a high-level, largely declarative language that is also based on our characterization of science and provides easy expression of the four classes of scientific activities in relation to R-domains. MDBL permits the easy integration of modules that support scientific modeling and database activities. We are currently applying the data model and languages to several EOS earth-science applications.

# 1 Introduction

High levels of abstraction are of increasing significance for many areas of application in which computational methods play an important role. In particular, we believe that there is an increasing need for such abstraction in languages that support modeling and database activities in many areas of application [11, 22]. Our justification for this need is that as applications become evermore complex, it is important to hide *irrelevant* computational details from the attention of individuals whose concern is with the application. Based upon our research interactions with a large number of earth scientists, we estimate that *many scientific teams focus up to 50% of their attention on computational issues that are irrelevant to their scientific research*. It is therefore clear that significant increases in efficiency are possible for applications in which computation is a major activity. Since these users may then focus their resources on the solution of the domain problems, rather than on "irrelevant" computational issues, we believe that the *efficiency* of such individuals, measured in terms of how quickly they find appropriate solutions to problems in their domains of application, should increase markedly.

In this paper, we present a conceptual data model and a high-level database and modeling language that provide the basis for the appropriate computational support, which we term a *modeling and database system* (MDBS) [20]. The data model and languages that we describe and exemplify are the result of long-term research collaborations with a number of earth scientists who are involved in NASA's Earth Observing System (EOS) program [7, 6]. The general goals of these investigators include the development and application of complex, spatially-distributed models of a variety of earth science phenomena. We have found that these investigators, in common with many other applications scientists, lack adequate computational support for the *iterative* construction and testing of their models. The generation of research results is heavily and negatively impacted by both the large size and processing requirements of these models, which are frequently coupled, and the need to develop such models using a large and heterogeneous collection of datasets. Such modeling typically requires the definition and manipulation of complex spatio-temporal entities.

The researchers also lack genuine *database support* for managing their fast growing collection of spatio-temporal and other datasets, which contain information in the form of satellite images and a large collection of digital datasets for elevation, vegetation, geology, soils, meteorology and hydrology. In large part, these data-handling difficulties result from the use of existing file systems as a repository for the datasets. The varied contents, formats, lineage, and the large size of datasets result in an unmanageable collection of files scattered over a network. The absence of database modeling and management further complicates the task of maintaining these datasets, which undergo continued growth and evolution. In addition, efficient access to the contents of these datasets is severely restricted in the current environment. Additional difficulties arise from the fact that while modeling and database activities are closely related from a scientific point of view, they are artificially separated by inadequate computational support. Furthermore, there is no computational support for coordinating the modeling and database activities of researchers involved in joint projects.

An MDBS should provide computational solutions to each of these problems. In particular, it should support the iterative development of scientific models in data-intensive and computationally-intensive applications; the construction, maintenance and access of scientific databases upon which such modeling activities are based; and a variety of important concepts, such as *models* and *multi-investigator projects*. Since a great proportion of scientific effort involves the definition and manipulation of *spatio-temporal entities*, and in particular with the construction, storage, accessing and manipulation of *spatio-temporal datasets*, it is critically important that much of the support of an MDBS be focused upon such entities.

Our strategy for designing and providing computational support to these researchers involves:

1. basing the design of an MDBS upon a simple model of scientific activity and upon a detailed investigation of the requirements of scientific investigators;

2. adopting a top-down design approach that is based on an appropriate high-level language that permits the simple expression of *much* of the functionality required by investigators;

3. designing a powerful data model for the organization of data and knowledge, and particularly data and knowledge about spatio-temporal domains, in a uniform manner that is tied, in particular, to our model of scientific activity;

4. providing a simple view of data in terms of datasets and not in terms of a distributed set of files;

5. providing a framework in which many scientific modeling tools can be easily integrated;

6. providing efficient support for:

    (a) exploratory data analysis

    (b) rapid prototyping and testing of models

    (c) production runs of models

Key aspects of this strategy involve the definition of appropriate, high-level languages in which scientists may express their activities in a manner that is "similar" to the language in which they express their science and the discovery of efficient computational support for such languages. A final but critical component of our strategy involves the testing of our ideas in terms of applying a prototype MDBS to solve *real* scientific problems whose solutions lie significantly beyond the computational support currently available to most earth scientists.

The focus of the current paper is upon an appropriate data model and an appropriate high-level language that are based upon a general characterization of scientific activity and that support the easy definition and manipulation of a large variety of complex spatio-temporal entities. The main contributions of the paper are an advanced and comprehensive data model and a high-level database and modeling language (MDBL). The data model is based on a simple characterization of scientific activity that captures the essence of scientific modeling and database activities in terms of four fundamental sets of activities. These activities are defined in relation to large lattices of *representational domains* (R-domains) that are employed to organize scientific knowledge, data and algorithms. The data model separates the specification and representation of R-domains of elements, and allows multiple representations that are explicitly controlled by the user. Datasets and transformations are organized in terms of an extensible lattice of abstract and concrete R-domains. In particular, the data model allows the easy definition and manipulation of large numbers of domains of spatio-temporal elements. We present many examples of such domains from our applications. MDBL is a high-level, largely declarative language that is also based on our characterization of science and provides easy expression of the four classes of scientific activities in relation to R-domains. MDBL permits the easy integration of modules that support scientific modeling and database activities.

This paper is organized as follows. A simple characterization of scientific research activity, especially involving intensive computations, is presented in Section 2. We base our data model upon this characterization. In Section 3 provides both an introduction to the data model and a detailed formal specification, together with examples to illustrate the concepts. Section 4 exemplifies the data model in terms of a variety of spatio-temporal domains. Sections 5 and 6 focus on a language for the expression of queries and computations and a simple high-level language for manipulating domains and transformations. Current implementation activities are briefly discussed in Section 7 and conclusions are presented in Section 8.

# 2 A Characterization of Scientific Activity as a Basis for Data Models and Languages

We believe it essential to base the design of an MDBS upon an appropriate characterization of the goals and activities of scientists, and in particular upon the idea that *the* fundamental goal of science is the discovery and investigation of symbolic models of phenomena. Both the data model and languages for an MDBS should be closely related to such a characterization. We briefly describe a characterization that we have developed in collaboration with earth science researchers [20, 22] and upon which we have based the data model and languages described below.

Of central importance for the success of scientific modeling activities, we believe, is the construction of appropriate sets of concepts, and, perhaps even more importantly, *appropriate representations* of those concepts in some language. We believe that much of the power of scientific investigations derives from the discovery and application of large numbers of clever and powerful representational domains and associated transformations. We therefore conceptualize scientific investigators as organizing much of their knowledge about the world in terms of large numbers of *domains of representations of entities* and *transformations between such domains.* In particular, we have found it convenient to differentiate between *conceptual domains* (C-domains), which relate to *abstract views* of entities and transformations, and *symbolic representations* of these abstractions, which we term *representational domains* (R-domains). In general, we may view a set of R-domains as being necessary for any computations that involve domain elements and transformations. For example, corresponding to the notion of a C-domain of *Polygons* and the associated transformations that provide semantics to the concept of polygons, there are a variety of R-domains whose elements may be interpreted as concrete representations of polygons and associated transformations, including representations based on sequences of *Points*, sequences of *Line-segments* and sets of *Half-planes*.

We may therefore characterize scientific investigations as activities in which scientists:

1. construct, evaluate and employ a large set of R-domains for representing conceptual entities;

2. construct, evaluate and employ representations of many transformations between these R-domains;

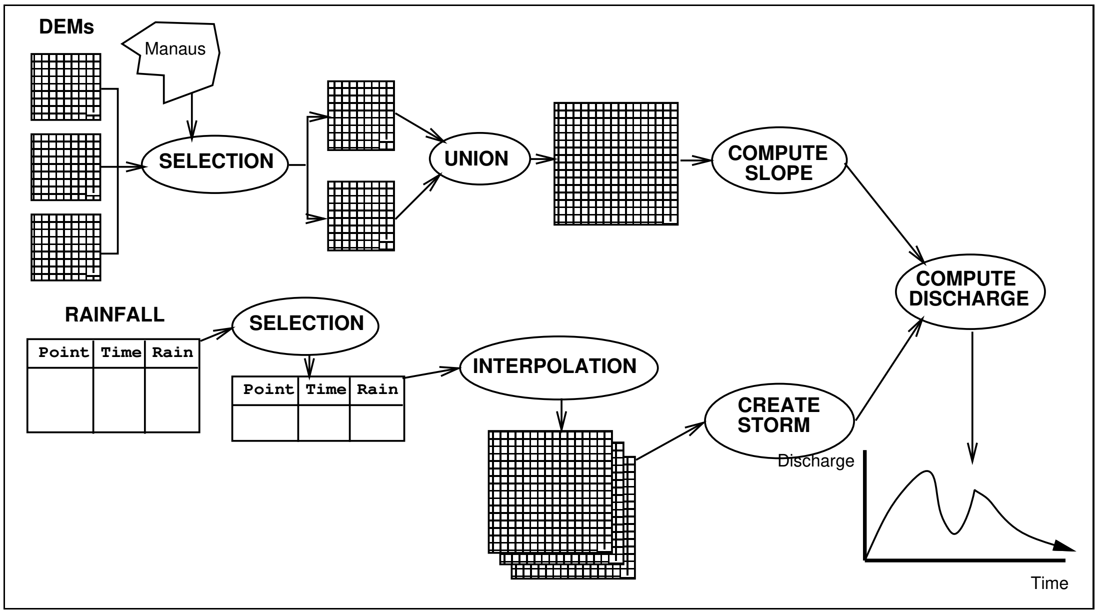


Figure 1: Computing Discharge: A Simplified Set of Operations

3. construct instances of R-domain elements;

4. apply sequences of specific transformations to specific sets of R-domain elements.

In terms of this viewpoint, we may characterize a *scientific model* as including a set of domains, a set of associated transformations, a set of domain instances and sequences of applications of transformations to domain instances.

We may illustrate some of these ideas in terms of an application problem which we are currently investigating in collaboration with a group of EOS investigators. One of the basic problems with which these earth scientists are faced is to construct models which represent the routing of water, sediments and solutes down the whole of the Amazon watershed. This problem is relatively complicated in terms of the various domains and transformations that are required for an adequate analysis. In order to exemplify the applicability of our characterization of scientific activity, we now briefly describe a *highly simplified and schematized view* of a subset of the domains, domain elements and transformations that have arisen in this application. We employ this example and its set of domains for illustrative purposes in the remainder of the paper.

In Figure 1, we show datasets and operations that provide a simplified view of the problem. Each of the datasets corresponds to an instance in some concrete R-domain and the operations correspond to transformations on R-domain elements. In Figure 2, we exemplify a sequence of the four classes of actions described above that accomplish the computation represented in Figure 1. In this figure, we represent these commands in natural language, while in Appendix II, we provide their translation into our high-level modeling and database language MDBL. It is to be stressed that expressing these commands in MDBL requires little more than an appropriate set of domains, domain instances, transformations. For example, one requires R-domains for digital elevation models (*DEMs*) and for hydrographs (*Hydrographs*) and their associated transformations.

In brief, the sequence of the computations involves first choosing *DEMs* of interest by intersecting them with an appropriate area and then combining them into one single DEM. Rainfall

1. Get the DEMs that intersect the Manaus area.
2. Mosaic this set of DEMs and produce a single resulting DEM.
3. Show a 3D display of this DEM.
4. Create a general class of maps that represent the slope function defined over a DEM.
5. Create the appropriate slope map from new the DEM constructed for the Manaus area and store it as a member of the general class just constructed.
6. Create a general class of maps that represent the equal travel distance for water flow over a given slope map.
7. Create the appropriate equal travel distance map from the slope entity just created for the Manaus dems and store it as a member of the general class of maps just constructed.
8. Find time-slices of rainfall from hour 1 to hour 12 on January 21 1989 for rain-fall records within the Manaus area.
9. Create a rainfall raster for each of the previous datasets using a rainfall interpolation routine.
10. Create a general class of raster representations of rainstorms.
11. Find and store all the rainstorm rasters that can be found in the rainfall raster just created.
12. Compute the hydrograph that would occur if this storm were to fall on the equal distance travel surface constructed above.
13. Display a plot of this hydrograph.

Figure 2: Computing Discharge: A Simplified Sequence of Actions

data from points inside this area and for certain specific time periods are retrieved and interpolated over the DEM. A transformation, *find-storms* is employed to extract all *Storms* from the rainfall data for each hour. The DEM is used to generate an isochrone "map" that models the flow of water from each point in the DEM to the mouth of the river. The *Hydrograph* is a representation of the discharge of water at the mouth as a function of time. It should be noted that the sequence of commands involves the creation of new domains, as well as the application of transformations such as *display* to domain elements.

We and our collaborators are strongly convinced that our characterization of science provides a strong conceptual basis on which to construct an MDBS. The data model and languages of the MDBS should support the definition and application of domains of elements and their associated transformations, and in particular of domains of entities that incorporate space and time in some essential manner. The data model and languages that we describe in the next section provide specific support for the four general classes of activity listed and exemplified above. Another component of MDBS that is important from the point of view of applications is a high-level, largely declarative modeling and database language (MDBL). This language may be employed to express the construction and application of an extensible and potentially *very* large collection of R-domains and associated transformations as well as the application of transformations to domain elements.

## 3 A Data Model for Domains and Transformations

We now focus on the construction of a data model that incorporates our basic notion of R-domains of symbolic representations. While the data model adopts many of the features that are found in object-oriented databases, such as type constructors and behavior encapsulation, its novelty lies in extensions that support the fundamental notion of R-domains. In summary, we represent R-domains in terms of a single abstract specification and many isomorphic concrete representations. Transformations are an integral part of R-domains, and possess an abstract specification and concrete representations. R-domains form a natural lattice that allows the inheritance of both concrete representations and transformations by sub-domains from super-domains. New domains may be created by explicit construction using type constructors and transformation languages or may be created by applying constraints to the elements of exiting domains (derived domains). Although the derived domains are not included in the formal presentation (the extension is not difficult), derived domains are shown in examples in Section 4, in which a variety of spatial and temporal domains are presented. Domains, domain elements an transformations may be defined using the language MDBL, which is defined in Section 6. The examples and the languages show both the simplicity and the expressiveness of the data model.

We first present an informal discussion of the structure of R-domains in §3.1. The data model is then formally introduced in §3.2 and §3.3.

### 3.1 Abstract and Concrete R-Domains

In our model, an R-domain has two components, namely an *abstract* part, and a *concrete* part. The *abstract* part or *abstract* R-domain consists of a set of attributes and transformations with signatures (or *methods* with types in object-oriented databases). Abstract domains are high-level specifications of conceptual domains, analogous to module interfaces in programming languages or the specification part of Abstract Data Types (ADTs). The concrete part consists of a set of *concrete* R-domains, each of which fully implements the abstract domain and all associated transformations.

We note two important facts concerning concrete domains. First, there may be more than one representation in a concrete domain that corresponds to a given element in the associated abstract domain. Hence it must be possible to map elements in a concrete domain into their appropriate *equivalence classes*. Consider, for example, the R-domain of *SimplePolygons* in the plane together with associated transformations relating to such properties as area, centroid and boundary. An associated concrete domain is *SimplePolygons⋆Points (SP⋆P)*, in which any simple planar polygon is represented in terms of some sequence $\langle p_1, \ldots, p_n \rangle$ of elements in the R-domain *Points*, together with associated constraints concerning the intersections of the linesegments that connect adjacent points in the sequence. It is clear that a given sequence of points does not provide a unique representation of a polygon and that any cyclic permutation of the sequence represents the same polygon. Hence we require transformations on the concrete domain that may be employed to determine whether any two elements represent the same element in the corresponding abstract domain (element equality). Second, there will in general be more than one concrete domain for a given abstract domain. A set of such concrete domains should be "equivalent" in some sense and they should map on to each other in an appropriate manner. In relation to *SimplePolygons*, for example, we may construct a second concrete domain *SimplePolygon⋆Linesegments (SP⋆L)* in which any simple planar polygon may be represented in terms of a sequence $\langle l_1, \ldots, l_n \rangle$ of elements from some representation of the domain *Linesegments*, together with constraints on the intersections and joining of pairs of linesegments in the sequence. It is clear that the concrete domains *SP⋆P* and *SP⋆L* are equivalent in terms of their relationship to the abstract domain *SimplePolygons* and that there is an isomorphic transformation between the equivalence classes of these two concrete domains and the associated set of transformations.

Given these observations, a general R-domain $D$ may be viewed as containing a set of elements together with:

1. a structurally isomorphic representation for each of the elements of its various concrete domains;
2. a representation of a (possibly empty) set of constraints on their values;
3. a representation of an associated set of transformations of the form $\rho : D \times D_1 \cdots \times D_n \rightarrow D'$;
4. a representation of the transformation that maps equivalent elements into each other, or in other words, an implementation for the *equality* predicate over the elements in the abstract domain;
5. a representation of the transformations that map concrete domains into equivalent concrete domains.

A general illustration of this structure of is shown in Figure 3, and a specific illustration is presented in the following example.

**Example 3.1:** Consider the domain *Points* of two dimensional points with a function *distance*. Its abstract representation is simply defined by a name and a function:

| | |
|---|---|
| **abstract domain** | *Points* |
| **transformation** | *distance* : $[\mathit{Points}, \mathit{Points}] \rightarrow \mathit{Reals}$ |

while two concrete representations may defined in terms of Cartesian and polar coordinates, with isomorphic mappings between the two representations. The following example illustrates the Cartesian concrete domain. A more complete definition is presented in Section 4.

| | |
|---|---|
| **concrete domain** | *Points⋆Cartesian (CP)* |
| **implements** | *Points* |
| **structure** | $[\mathit{Reals}, \mathit{Reals}]$ |
| **transformation** | $\mathit{distance}((x, y), (x', y')) = \sqrt{(x - x')^2 + (y - y')^2}$ |

□

There are several advantages in separating abstract and concrete representations of domains. First, the elements from the same conceptual domain can be represented in different structural forms. It allows easier cooperation between databases and enhances efficiency, since enforcing uniform representations is unnecessary, sometimes undesired, and even impossible in the scientific applications. Second, new concrete domains may be added without restructuring existing concrete domains and hence the system is more flexible and extensible. The concept of concrete domains is similar to, but different from, the concept of ADTs in the programming language context. It is similar in the sense of providing abstraction while hiding irrelevant details; it is different in the sense that an ADT usually has one representation, while an abstract domain may have

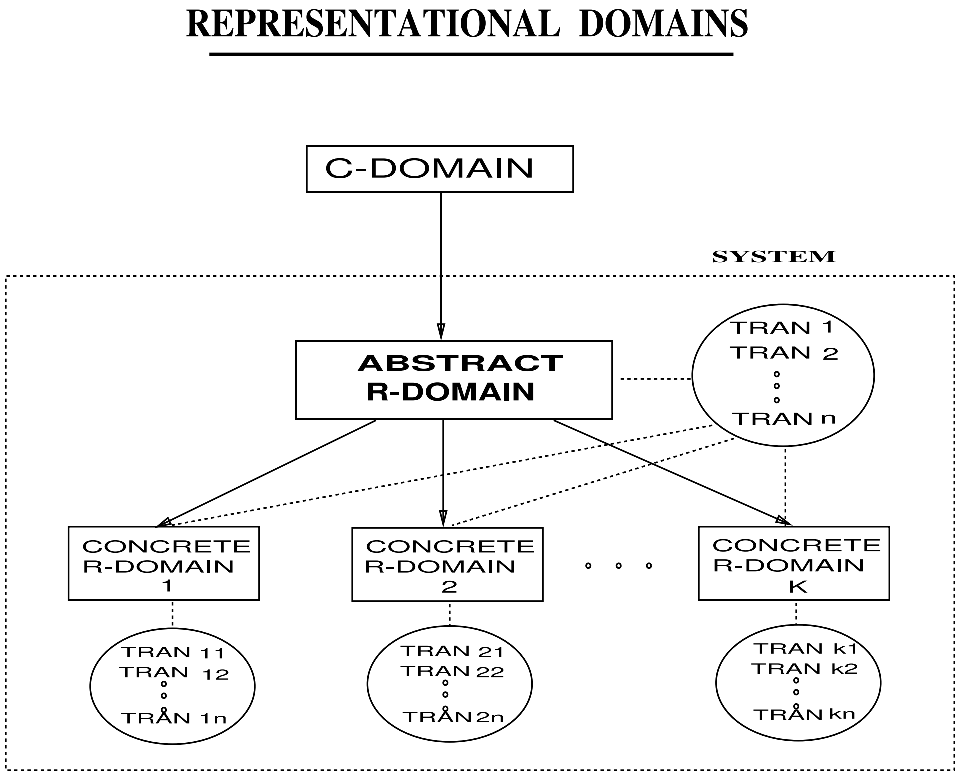


Figure 3: An abstract domain may have $n$ concrete domains

several. Furthermore, while an ADT in the programming language context hides the representation, different concrete domains in the database context are made explicitly available to the user, since different representations are extremely important in many scientific applications. Interestingly, the abstract domains can be viewed as "polymorphic" types. The main difficulty in allowing different representations is to maintain the consistency of such differently structured elements, especially under updates.

## 3.2 A Formal Data Model

We now provide a formal description of our data model for R-domains and transformations in scientific databases. The model extends many existing object-oriented models, such as $O_2$, with the idea of a "federation" of concrete representations for conceptual domains. We introduce the notion of "type", "object equivalence", and "structural databases" in this section and incorporate "transformations" and "constraints" in the following subsection.

To begin with, we assume the existence of the following pairwise disjoint sets:

- a countably infinite set $\mathcal{B}$ of *basic domain names* (e.g., *Reals*, *Integers*, *Strings*, *Boolean*, etc.);
- a countable set $\mathcal{V}(D)$ of *atomic values* for each $D \in \mathcal{B}$;
- a countably infinite set $\mathcal{D}$ of *domain names;*

- a countably infinite set $\mathcal{T}$ of *transformation names;*
- a countably infinite set $\mathcal{O}$ of *object identifiers (oid's).*

We assume that the basic domains $\mathcal{B}$ also includes *integer range types*, denoted as $(0..n)$ or simply $(n)$, which has a set of values $\{0, \ldots, n\}$. We allow tuple, set, and finite function constructors to build complex "types". More formally,

**Definition:** The family of *types* is recursively defined as follows:

- $D$ is a *basic* type if $D \in \mathcal{B}$;
- $D$ is an *object* type if $D \in \mathcal{D}$;
- $[\tau_1, \ldots, \tau_n]$ is an $n$-ary *tuple* type if $n \geq 0$ and for each $i \in [1..n]$, $\tau_i$ is a type;
- $\{\tau\}$ is a *set* type if $\tau$ is a type;
- $\tau \rightarrow \tau'$ is a *finite (stored) function* type if $\tau$, $\tau'$ are types;

Let $\boldsymbol{\tau}$ denote the family of all types. For each $D \in \mathcal{D}$ and each $\tau \in \boldsymbol{\tau}$, $\tau$ *uses* $D$ if $D$ occurs in $\tau$.

For example, $\tau_{\mathit{Points}} = [\mathit{Reals}, \mathit{Reals}]$ is a binary tuple type. We can use $\tau_{\mathit{Points}}$ for representing two dimensional points. A two dimensional spatial mapping can be represented as a finite function from a set of points to real numbers (representing elevation,, rainfall, etc.), i.e., as an element of the type $\tau_{\mathit{2Dmap}} = \tau_{\mathit{Points}} \rightarrow \mathit{Reals}$.

In practice, arrays are very useful data structures. Since an array type can be defined using function and integer range types, we do not include the array type constructor explicitly. For example, an array type of the following form:

- $\mathit{Reals}((99), (99))$ (the type of two dimensional arrays of reals)

can be defined as the function type: $[(99), (99)] \rightarrow \mathit{Reals}$. Also, sequence types can be represented using function types with constraints on the domains. In informal discussions, we will also use $\langle e_1, \ldots, e_n \rangle$ to represent a sequence of $n$ elements.

Given a type $\tau$, the elements or "domain" of the type is given by the next definition. An *object assignment* is a total mapping $\pi$ from $\mathcal{D}$ to $powerset(\mathcal{O})$. An object assignment $\pi$ is *universal* if $\pi(D) = \mathcal{O}$ for each $D \in \mathcal{D}$. The universal object assignment is denoted as $\widehat{\pi}$.

**Definition:** Let $\pi$ be an object assignment. The *(object) domains* of types $\tau$ with respect to $\pi$, denoted by $\mathcal{V}_\pi(\tau)$, are recursively defined as follows. ($\mathcal{V}_{\widehat{\pi}}(\tau)$ is simply denoted as $\mathcal{V}(\tau)$.)

- If $\tau = D$ is a basic type, then $\mathcal{V}_\pi(\tau) = \mathcal{V}(D)$;
- If $\tau = D$ is an object type, then $\mathcal{V}_\pi(\tau) = \pi(D)$;
- If $\tau = [\tau_1, \ldots, \tau_n]$ is a tuple type, then $\mathcal{V}_\pi(\tau) = \mathcal{V}_\pi(\tau_1) \times \cdots \times \mathcal{V}_\pi(\tau_n)$;
- If $\tau = \{\tau'\}$ is a set type, then $\mathcal{V}_\pi(\tau) = \mathit{powerset}^{\mathrm{finite}}(\mathcal{V}_\pi(\tau'))$;

- If $\tau = \tau' \rightarrow \tau''$ is a finite function type, then $\mathcal{V}_\pi(\tau)$ is the set of all partial functions from $\mathcal{V}_\pi(\tau')$ to $\mathcal{V}_\pi(\tau'')$ with finite domains.

Each member of $\mathcal{V}_\pi(\tau)$ is a(n *object*) *value* of type $\tau$ under $\pi$. Each finite subset of $\mathcal{V}_\pi(\tau)$ is an instance of type $\tau$ under $\pi$. The family of instances of type $\tau$ under $\pi$ is denoted as $inst_\pi(\tau)$.

**Example 3.2:** Suppose *Points* $\in \mathcal{D}$ is the name for the domain of two dimensional points. Let $\tau_{Points}$ denote the type of domain *Points*, and $\tau_{map}$ be the function type *Points* $\rightarrow$ *Reals* representing the spatial maps. Suppose $\pi$ maps *Points* to the set $\{o_1, ..., o_4\}$ Then the following is a value for $\tau_{map}$:

$$\{(o_1 \rightarrow 1.5), (o_2 \rightarrow 1.0), (o_3 \rightarrow 1.4), (o_4 \rightarrow 0.7)\}$$

$\square$

Since the data model allows inheritance of transformations, the types of super/sub-domains, however, must satisfy the "subtyping" requirements. Formally, for the set $\boldsymbol{\mathcal{T}}$ of all types, the binary *subtyping* relation ($\preceq$) can be defined by the following rules.

1. $\tau \preceq \tau$ for each $\tau \in \boldsymbol{\mathcal{T}}$;
2. $[\tau_1, \ldots, \tau_n] \preceq [\tau'_1, \ldots, \tau'_m]$ if $n \geq m$ and for each $i \in [1..m]$, $\tau_i \preceq \tau'_i$;
3. $\{\tau\} \preceq \{\tau'\}$ if $\tau \preceq \tau'$;
4. $\tau_1 \rightarrow \tau_2 \preceq \tau'_1 \rightarrow \tau'_2$ if $\tau_1 \preceq \tau'_1$ and $\tau'_2 \preceq \tau_2$.

Intuitively, types are used to represent elements in concrete domains, that is, each concrete domain has an associated type. As noted above, however, a domain element may often be represented by more than one value of the type. In our model, an object has a unique identifier and a value of an appropriate type. The equivalence relations in fact define equalities on objects or object identifiers with respect to certain domains (e.g. polygon). Such equivalence relations are extensions of the notion of "key" or "surrogate" in relational and semantic databases [24, 9].

To incorporate such equivalence relations in a formal manner, we introduce the following concepts. Suppose $\tau \in \boldsymbol{\mathcal{T}}$ is a type. A binary relation $E_\tau$ over $\mathcal{V}(\tau)$ is a *candidate equality for* $\tau$ if it is an equivalence relation and for each pair of elements $u, v$ in $\mathcal{V}(\tau)$, $E_\tau(u, v)$ implies:

1. the set of oid's occurring in $u$ = the set of oid's occurring in $v$; and
2. the set of atomic values occurring in $u$ = the set of atomic values occurring in $v$.

For example, let $\rho$ be a set of these "cyclic" permutations. A candidate equality $\simeq_{\langle\rangle_\rho}$ can be defined based on $\rho$ and used as the equality for the polygon concrete domain. In general, the candidate equalities must be consistent with the inheritance hierarchy. To define formally the notion of consistency, we introduce the notions of "restriction", "extension" of values to a supertype, subtype (respectively).

Let $\sigma, \tau$ be two types in $\boldsymbol{\mathcal{T}}$ such that $\sigma \preceq \tau$. The *restriction* of values $u$ in $\mathcal{V}(\sigma)$ to $\tau$ denoted as $u|_\tau$, and the *extension* of values $v$ in $\mathcal{V}(\tau)$ to $\sigma$, denoted as $v|^\sigma$, are defined recursively as follows:

- If $\sigma = \tau$, then $u|_\tau = \{u\}$ and $v|^\sigma = \{v\}$;
- If $\sigma = [\sigma_1, \ldots, \sigma_n]$, $\tau = [\tau_1, \ldots, \tau_m]$ are tuple types and $u = [u_1, \ldots, u_n]$, $v = [v_1, \ldots, v_m]$, then

$$u|_\tau = \{[u'_1, \ldots, u'_m] \mid \forall i \in [1..m], u'_i \in u|_{\tau_i}\},$$
$$v|^\sigma = \{[v'_1, \ldots, v'_n] \mid \forall i \in [1..m], u'_i \in u|^{\sigma_i}, \forall i \in [(m+1)..n], u'_i \in \mathcal{V}(\sigma_i)\};$$

- If $\sigma = \{\sigma'\}$, $\tau = \{\tau'\}$ are set types and $u = \{u_1, \ldots, u_n\}$, $v = \{v_1, \ldots, v_m\}$, then

$$u|_\tau = \{\{u'_1, \ldots, u'_n\} \mid \forall i \in [1..n], u'_i \in u_i|_{\tau'}\},$$
$$v|^\sigma = \{\{v'_1, \ldots, v'_m\} \mid \forall i \in [1..m], v'_i \in v_i|^{\sigma'}\};$$

- If $\sigma = \sigma' \rightarrow \sigma''$, $\tau = \tau' \rightarrow \tau''$ are finite function types and $u = \{(u_1 \rightarrow u'_1), \ldots, (u_n \rightarrow u'_n)\}$, $v = \{(u_1 \rightarrow u'_1), \ldots, (u_n \rightarrow u'_n)\}$, then

$$u|_\tau = \{\{(u''_1 \rightarrow u'''_1), \ldots, (u''_n \rightarrow u'''_n)\} \in \mathcal{V}(\tau' \rightarrow \tau'') \mid \forall i \in [1..n], u''_i \in u_i|_{\tau'}, u'''_i \in u'_i|^{\tau''}\},$$
$$v|^\sigma = \{\{(v''_1 \rightarrow v'''_1), \ldots, (v''_n \rightarrow v'''_n)\} \in \mathcal{V}(\sigma' \rightarrow \sigma'') \mid \forall i \in [1..n], v''_i \in v_i|^{\sigma'}, v'''_i \in v'_i|_{\sigma''}\}.$$

Suppose $\sigma, \tau$ are types such that $\sigma \preceq \tau$ and $E_\sigma$, $E_\tau$ are their candidate equalities (respectively). $E_\sigma$ and $E_\tau$ are *consistent* if for each $u, v \in \mathcal{V}(\sigma)$, $s, t \in \mathcal{V}(\tau)$ such that $E_\sigma(u, v)$ and $E_\tau(s, t)$, the following are true:

1. For each $u' \in u|_\tau, v' \in v|_\tau$, $E_\tau(u', v')$; and
2. For each $s' \in s|^\sigma, t' \in t|^\sigma$, $E_\sigma(s', t')$.

Finally, we associate types with transformations. Intuitively speaking, each transformation is viewed as defining a function.

**Definition:** A *transformation signature* is an expression $t : [\tau_1, \ldots, \tau_n] \rightarrow \tau$, where $t \in \mathcal{T}$ is a transformation name and $\{\tau, \tau_1, \ldots, \tau_n\} \subseteq \boldsymbol{\mathcal{T}}$.

We may now define the *structural schema* of databases. Intuitively, a structural schema includes a set of abstract domain names and a set concrete domain names; a mapping specifying the corresponding abstract domain for each concrete domain; and a type assignment defining a type for each concrete domain. The object domains used in types are abstract domains. The object equality for each concrete domain is also provided. Each abstract domain includes a set of transformations with signatures. Isomorphisms between different concrete domains of each abstract domain are also provided. Finally, a lattice may be defined over abstract domains in terms of a binary sub-domain relation. As in many object-oriented models, sub-domains inherit all transformations of a super-domain, while the structure of each concrete domain of the super-domain is also inherited by some concrete domain of the sub-domain. The equality relations defined on the two concrete domains are "consistent".

**Definition:** A *database structural schema* is a system $\boldsymbol{D} = (\boldsymbol{A}, \boldsymbol{C}, A, T, \simeq, \textbf{isa}, M, H)$, where

1. $\boldsymbol{A}$ and $\boldsymbol{C}$ are two finite, disjoint subsets of $\mathcal{D}$ (representing abstract and concrete domains, respectively);

2. $A$ is a total mapping from $\boldsymbol{C}$ to $\boldsymbol{A}$;

3. $T$ is a total mapping from $\boldsymbol{C}$ to the set $\boldsymbol{\tau}$ of types such that for each $D \in \boldsymbol{C}$ and each $D' \in \mathcal{D}$, if $T(D)$ uses $D'$ then $D' \in \boldsymbol{A}$ (for notational convenience, we use $T_D$ to denote $T(D)$);

4. $\simeq$ is a mapping from $\boldsymbol{C}$ such that for each $D \in \boldsymbol{C}$, $\simeq_D$ is a candidate equality for $T_D$;

5. **isa** $\subseteq \boldsymbol{A} \times \boldsymbol{A}$ is a binary relation satisfying the following conditions:

   (a) **isa** is reflexive;

   (b) **isa** is acyclic; and

   (c) If[1] $D_1$ **isa** $D_2$ for some $D_1, D_2 \in \boldsymbol{A}$, then for each concrete domain $D'_2$ of $D_2$, i.e. $D'_2 \in \boldsymbol{C}$ with $A(D'_2) = D_2$, there exists a concrete domain $D'_1 \in \boldsymbol{C}$ satisfying

      i. $A(D'_1) = D_1$, and

      ii. $T_{D'_1}$ is a subtype of $T_{D'_2}$, i.e., $T_{D'_1} \preceq T_{D'_2}$;

      iii. $\simeq_{D'_1}$ and $\simeq_{D'_2}$ are consistent.

   The transitive closure of **isa** is denoted as **isa***;

6. $M$ is a mapping from $\boldsymbol{A}$ to the set of all finite sets of transformation signatures with respect to $\boldsymbol{A}$ such that

   (a) For each $D \in \boldsymbol{A}$, each signature $t : [\tau_1, \ldots, \tau_n] \to \tau_0$ in $M(D)$,

      i. $n \geq 1$;

      ii. $\tau_1 = D$;

      iii. For each $D' \in \mathcal{D}$, and each $i \in [0..n]$, if $\tau_i$ uses $D'$, then $D' \in \boldsymbol{A}$;

   (b) For each pair of distinct $D, D' \in \boldsymbol{A}$, the transformation names of the signatures in $M(D)$ are disjoint from that of the signatures in $M(D')$;

7. $H$ is a mapping from the set $\{(D, D') \mid D, D' \in \boldsymbol{C}, A(D) = A(D')\}$ to functions such that $H(D, D')$ is a total function from $\mathcal{V}(T_D)$ to $\mathcal{V}(T_{D'})$. (We also denote $H(D, D')$ simply as $H_{D \to D'}$.) $H$ must satisfy the following conditions:

   (a) If $v = H_{D \to D'}(u)$, then $u$ and $v$ use the same set of oid's;

   (b) $u \simeq_D v$ implies $H_{D \to D'}(u) \simeq_{D'} H_{D \to D'}(v)$ for every $u, v \in \mathcal{V}(T_D)$; and

   (c) $u \simeq_D H_{D \to D'}(H_{D \to D'}(u))$ for each $u \in \mathcal{V}(T_D)$.

[1] We use infix notation for the binary relation **isa**.

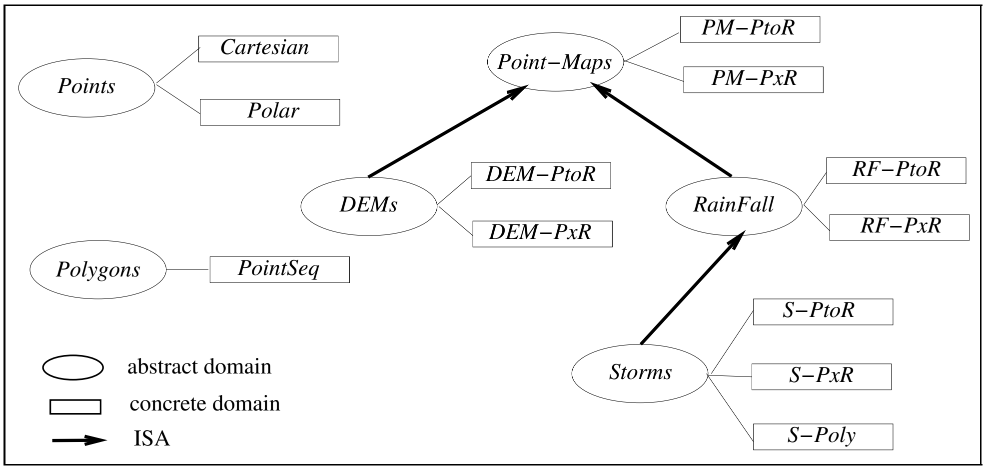


Figure 4: Abstract and Concrete Domains

**Example 3.3:** Consider an application with following domains: *Points*, *Polygons*, *Point-Map*, *DEM*, *RainFalls*, and *Storms*, where *Point-Maps* has sub-domains *DEMs* and *RainFalls* and *RainFalls* has sub-domain *Storms*. *Point-Maps* has two concrete representations: (1) as a function from *Points* to *Reals*, and (2) as a binary relation of *Points* and *Reals*. These representations are inherited by all sub-domains. The *Storms* domain has an additional representation using domain *Polygons*. The domains are shown in Figure 4, where ovals represent abstract domains and rectangles represent concrete domains.

The types of the concrete domains, $T_D$ for domain $D$, are listed as follows. [2]

- $T_{\mathit{Cartesian}} = [\mathit{Reals}, \mathit{Reals}]$;
- $T_{\mathit{Polar}} = [\mathit{Reals}, \mathit{Reals}]$;
- $T_{\mathit{PointSeq}} = \mathit{Integers} \rightarrow \mathit{Points}$;
- $T_{\mathit{PM\text{-}PtoR}} = T_{\mathit{DEM\text{-}PtoR}} = \mathit{Points} \rightarrow \mathit{Reals}$, $T_{\mathit{RF\text{-}PtoR}} = T_{\mathit{S\text{-}PtoR}} = [\mathit{Points} \rightarrow \mathit{Reals}, \mathit{TimePoints}]$;
- $T_{\mathit{PM\text{-}PxR}} = T_{\mathit{DEM\text{-}PxR}} = \{[\mathit{Point}, \mathit{Real}]\}$ and $T_{\mathit{RF\text{-}PxR}} = T_{\mathit{S\text{-}PxR}} = \{[\mathit{Points}, \mathit{Reals}, \mathit{TimePoints}]\}$;
- $T_{\mathit{S\text{-}Poly}} = [\mathit{Polygons}, \mathit{Points} \rightarrow \mathit{Reals}, \mathit{TimePoints}]$.

The transformations for these domains are listed in the following.

| | |
|---|---|
| *Points* | *distance*: $[\mathit{Points}, \mathit{Points}] \rightarrow \mathit{Reals}$ |
| *Polygons* | *area*: $\mathit{Polygons} \rightarrow \mathit{Reals}$ |
| *DEMs* | *compute-discharge*: $[\mathit{DEMs}, \mathit{Storms}] \rightarrow \{[\mathit{TimePoints}, \mathit{Reals}]\}$ |
| *RainFalls* | *find-storms*: $\mathit{RainFalls} \rightarrow \{\mathit{Storms}\}$ |

The structural database schema hence includes the abstract and concrete domains, the types for concrete domains, the subdomain relation shown above, and also appropriate equalities for, and isomorphic mappings between, concrete domains. □

[2]We also view a type $\tau$ as a unary tuple type $[\tau]$.

We now define instances for structural schemas $(\boldsymbol{A}, \boldsymbol{C}, A, T, \simeq, \mathbf{isa}, M, H)$. The mapping $A$ in the schema is extended to $\boldsymbol{A} \cup \boldsymbol{C}$ such that it is the identity mapping on $\boldsymbol{A}$. Intuitively, in a database structural instance, each domain is assigned a set of objects (or object identifiers) where domains for distinct abstract R-domains have disjoint sets of objects and those for the same abstract domain have the same set of objects. Since objects are represented by different structures, the instance also assigns several values to each object. Finally, the set of objects for each domain is the union of all objects in its descendant domains.

**Definition:** Let $\boldsymbol{D} = (\boldsymbol{A}, \boldsymbol{C}, A, T, \simeq, \mathbf{isa}, M, H)$ be a structural schema. A *(database) structural instance* is a pair $\boldsymbol{d} = (\pi, \delta)$ where

1. $\pi$ is an object assignment such that
   (a) for each $D \notin \boldsymbol{A} \cup \boldsymbol{C}$, $\pi(D) = \emptyset$;
   (b) for each $D \in \boldsymbol{A} \cup \boldsymbol{C}$, $\pi(D)$ is finite and $\pi(D) = \pi(A(D))$; and
   (c) for each $D, D' \in \boldsymbol{A} \cup \boldsymbol{C}$, $A(D) \neq A(D')$ implies $\pi(D) \cap \pi(D') = \emptyset$.

   We denote $\pi^*(D) = \cup\{\pi(D') \mid A(D') \ \mathbf{isa}^* \ A(D)\}$.

2. $\delta$ is a partial mapping from $\boldsymbol{C} \times \bigcup_{D \in \boldsymbol{C}} \pi(D)$ such that for each $D \in \boldsymbol{C}$ and each $o \in \pi(D)$, $\delta(D, o) \in \mathcal{V}_\pi(T_D)$; $\delta$ is undefined elsewhere. The mapping $\delta$ also satisfies the following condition:

   - For each $D, D' \in \boldsymbol{C}$ having $A(D) = A(D')$ and each $o \in \pi(D)$,
   $$H_{D \to D'}(\delta(D, o)) \simeq_{D'} \delta(D', o).$$
   - For each $D \in \boldsymbol{C}$ and each $o, o' \in \pi(D)$, $\delta(D, o) \simeq_D \delta(D, o')$ implies $o = o'$.

The family of all instances of $\boldsymbol{D}$ is denoted as *inst*$(\boldsymbol{D})$.

**Example 3.4:** Consider the structural schema in Example 3.3. Part of a database instance is shown below.

$$\begin{array}{l}
\pi(\textit{Points}) = \{o_1, o_2, ..., o_{30}\} \\
\pi(\textit{Polygons}) = \{p_1, ..., p_5\} \\
\pi(\textit{Point-Maps}) = \{m_1\} \\
\pi(\textit{DEMs}) = \{m_2\} \\
\pi(\textit{RainFalls}) = \pi(\textit{RainFalls}) = \emptyset \\
\delta(\textit{Points}, o_1) = [1, 1] \\
\quad \cdots \\
\delta(\textit{Points}, o_30) = [250, 150] \\
\delta(\textit{Point-Maps}) = \{(o_1 \to 0.5), ..., (o_{25} \to 1.3)\} \\
\quad \cdots
\end{array}$$

□

### 3.3 Transformations and Integrity Constraints

We further let $\mathcal{I}$ be a countably infinite, disjoint set of *programs*. We view that $\mathcal{I}$ provides the actual implementation (executable code) for transformations. Informally, we define an *implementation* of transformations to be a mapping from transformation signatures to $\mathcal{I}$, satisfying typing restrictions.

Let $\boldsymbol{D} = (\boldsymbol{A}, \boldsymbol{C}, A, T, \simeq, \mathbf{isa}, M, H)$ be a structural schema. We denote $u \simeq_{D,D'} v$, where $D, D' \in \boldsymbol{C}$ such that $A(D) = A(D')$ and $u \in \mathcal{V}(T_D), v \in \mathcal{V}(T_{D'})$, if $H_{D \to D'}(u) \simeq_{D'} v$. It is easy to verify that

$$u \simeq_{D,D'} v \text{ iff } v \simeq_{D',D} u.$$

**Definition:** Suppose $\boldsymbol{D} = (\boldsymbol{A}, \boldsymbol{C}, A, T, \simeq, \mathbf{isa}, M, H)$ is a structural schema. An *implementation* is a mapping $I$ from $\cup_{D \in \boldsymbol{C}} \{D\} \times M(D)$ satisfying the typing requirements and the following condition:

- For each $D, D' \in \boldsymbol{C}$ that $A(D) = A(D')$, each transformation signature $\sigma \in M(A(D))$, and each $u \in \mathcal{V}(T_D), v \in \mathcal{V}(T_{D'})$,

  $$u \simeq_{D,D'} v \quad \text{implies} \quad \forall \vec{x}\, I(D, \sigma)(u, \vec{x}) \simeq_{D,D'} I(D', \sigma)(v, \vec{x})$$

  (We assume the output type of the transformation $\sigma$ is $A(D)$. For general output types, the condition can be written similarly.)

As in many other kinds of database applications, integrity constraints are an important part of databases. However, a distinction is the fact that the usual types of integrity constraints for conventional database applications are far from being useful in scientific databases. In this section, we define a many-sorted high order logic for the database structures introduced in the previous section and use it to express nontrivial constraints.

The logic further extends many languages for many extensions to the relational model. We assume that there exists a countably infinite set of *variables*, and that each variable has an implicitly associated type.

Let $\boldsymbol{D} = (\boldsymbol{A}, \boldsymbol{C}, A, T, \simeq, \mathbf{isa}, M, H)$ be a database structural schema. We now define the notion of terms in the language. The family of *terms* with respect to the schema $\boldsymbol{D}$ is defined by the following:

1. Values in $\mathcal{V}(D)$ where $D \in \mathcal{B}$ are terms;
2. Variables of types constructed from types $\mathcal{B} \cup \boldsymbol{A}$ are terms;
3. $\nu^D(x)$ is a term of type $T_D$ if $D \in \boldsymbol{C}$, $x$ is a variable of type $A(D)$;
4. $s.i$ is a term of the type $\tau_i$ if $s$ is a term of the tuple type $[\tau_1, \ldots, \tau_n]$ and $i \in [1..n]$;
5. $[s_1, \ldots, s_n]$ is a term of the tuple type $[\tau_1, \ldots, \tau_n]$ if $n \geq 0$ and for each $i \in [1..n]$, $s_i$ is a term of the type $\tau_i$;
6. $\{s_1, \ldots, s_n\}$ is a term of the type $\{\tau\}$ if $n \geq 0$, $s_1, \ldots, s_n$ are terms of the type $\tau$;

7. $\{(s_1 \rightarrow t_1), \ldots, (s_k \rightarrow t_k)\}$ is a term of the function type $\tau \rightarrow \tau'$ if $s_i$'s, $t_i$'s are terms of type $\tau$, $\tau'$ (respectively);

8. $s(s')$ is a term of type $\tau'$ if $s, s'$ are terms of types $\tau \rightarrow \tau'$, $\tau$ (respectively).

We also allow terms to construct integer ranges such as $[1..x]$ (where $x$ is an integer). Function terms in (8) of the above definition, include values of finite function types, transformations, arithmetic operators (such as $+, \times$, ..., log, sin, ..., etc.) and also the following structure-specific operations for sets and finite functions.

- For set terms $s, s'$ of type $\{\tau\}$:
  - $pset(s)$ returns the powerset of $s$;
  - $s \cup s'$, $s - s'$ return the set union and difference of $s, s'$ (respectively);
  - $clps(s)$, if $\tau = \{\tau'\}$ is a set type, returns the union of all elements (sets) in $s$.
- For finite function terms $s, s'$ of type $\tau \rightarrow \tau'$:
  - $dm(s)$, $rg(s)$ return the domain, range of the function $s$, which are of types $\{\tau\}$, $\{\tau'\}$ (respectively);
  - $s \uplus s'$ returns the function union of $s, s'$, defined as:

$$s \uplus s'(x) = \begin{cases} s(x) & \text{if } s(x) = s'(x) \\ \text{undefined} & \text{otherwise} \end{cases}$$

  - $s \Downarrow t$ (domain restriction), where $t$ is a term of type $\{\tau\}$, returns the function of $s$ restricted to the new domain $dm(s) \cap t$.
  - $s \Uparrow t$ (range restriction), where $t$ is a term of type $\{\tau'\}$, returns the function of $s$ restricted to the new range $rg(s) \cap t$, i.e.,

$$s \Uparrow t(x) = \begin{cases} s(x) & \text{if } s(x) \in t \\ \text{undefined} & \text{otherwise} \end{cases}$$

  - $s'' \circ s$, where $s''$ is a term of type $\tau' \rightarrow \tau''$, returns the composition of functions $s, s''$.

Since integer operations are also allowed on integer range types, we treat these range types as finite fields.

The family of *range-restricted well-formed formulas (rr-wff's or rr-formulas)* is now defined by the following.

- $s =_\tau s'$ is an *atomic* rr-formula if $s, s'$ are terms of type $\tau$ (when the context is clear, $=_\tau$ is simply written as $=$);
- $s \simeq_D s'$ is an *atomic* rr-formula if $s, s'$ are terms of type $T_D$;
- $s \in_{\tau,\tau'} s'$ is an *atomic* rr-formula if $s, s'$ are terms of the type $\tau, \tau'$ and $\tau' = \{\tau\}$ (when the context is clear, $\in_{\tau,\tau'}$ is simply written as $\in$);

- $D(s)$ is an *atomic* rr-formula if $D \in \boldsymbol{A}$, $s$ is a variable of type $D' \in \boldsymbol{A}$ and $D'$ $\mathbf{isa}^*$ $D$;
- $\neg\phi$, $\phi_1 \wedge \phi_2$, $\phi_1 \vee \phi_2$ are rr-formulas if $\phi, \phi_1, \phi_2$ are rr-formulas;
- $\forall x \in D\, \phi$ and $\exists x \in D\, \phi$ are rr-formulas if $x$ is a variable of type $D' \in \boldsymbol{A}$ and $D'$ $\mathbf{isa}^*$ $D$;
- $\forall x \in s\, \phi$ and $\exists x \in s\, \phi$ are rr-formulas if $x$ is a variable of type $\tau$, $s$ a term of type $\{\tau\}$, and $\phi$ is an rr-formula.

A *sentence* is an rr-formula without any free variables. In addition to the above rr-formulas, we also allow rr-formulas "$s(s') \downarrow$", where $s, s'$ are terms of types $\tau \to \tau'$ and $\tau$ (respectively), "$s$", where $s$ is a term of type *Booleans*, etc, as rr-formulas stating if $s(s')$ is defined, or if $s$ has the value *true*. Obviously, they are definable using other rr-formulas.

**Example 3.5:** Let $x$ be a variable of type $\{[\mathit{Points}, \mathit{Reals}]\}$ and $y_1, y_2$ two variables of types *Points*, *Reals* (respectively). The following rr-formula $\phi(x)$ states that $x$ really represents a mapping from *Points* to *Reals*, or in other words, a functional dependency from column 1 to 2 holds.

$$\phi(x) = \forall y_1 \in x\, \forall y_2 \in x\, (y_1.1 \simeq_{\mathit{Points}} y_2.1 \to y_1.2 = y_2.2)$$

Note that $\simeq_{\mathit{Points}}$ is the equality on *Points*. □

Let $\boldsymbol{D} = (\boldsymbol{A}, \boldsymbol{C}, A, T, \simeq, \mathbf{isa}, M, H)$ be a structural schema and $\boldsymbol{d} = (\pi, \delta)$ an instance of $\boldsymbol{D}$. An *assignment* (with respect to $\boldsymbol{d}$ ) is a mapping from variables to values of the corresponding types. Let $\rho$ be an assignment, we extend $\rho$ to general terms naturally as follows.

- If $s$ is a value, then $\rho(s) = s$ and $s$ is a variable, then $\rho(s) = \rho(s)$.
- If $\nu^D(x)$ is a term, then $\rho(\nu^D(x)) = \delta(D, x)$.
- If $s = [s_1, \ldots, s_n]$ is a tuple, $\rho(s.i) = \rho(s_i)$.
- $\rho([s_1, \ldots, s_n]) = [\rho(s_1), \ldots, \rho(s_n)]$.
- $\rho(\{s_1, \ldots, s_n\}) = \{\rho(s_1), \ldots, \rho(s_n)\}$.
- If $s = \{(s_1 \to t_1), \ldots, (s_k \to t_k)\}$ is a (finite) function term, let $f$ be the (possibly multi-valued) mapping defined in the following:

  $$f(\rho(s_1)) = \rho(t_i) \qquad \text{for each } i \in [1..k]$$

  Then, $\rho(s) = f$ if $f$ is a single-valued function, and $\rho(s)$ is undefined otherwise;
- If $s$ is a function, $\rho(s(s')) = \rho(s)(\rho(s'))$.

The assignments $\rho$ are extended to composed function terms with operations in a straightforward way. Now we define the semantics of the language.

**Definition:** Let $D$ be a database schema and $\boldsymbol{d} = (\pi, \delta)$ an instance of $D$. Suppose $\phi$ is an rr-formula and $\rho$ an (extended) mapping from variables to values of the corresponding type with respect to the database $\boldsymbol{d}$. *Satisfaction of $\phi$ under $\rho$ by* $\boldsymbol{d}$, denoted as $\boldsymbol{d} \models \phi[\rho]$, is defined inductively as follows.

- $\boldsymbol{d} \models s = s'[\rho]$ if $\rho(s)$ and $\rho(s')$ are defined and $\rho(s) = \rho(s')$;
- $\boldsymbol{d} \models s \simeq_D s'[\rho]$ if $\rho(s)$ and $\rho(s')$ are defined and $\rho(s) \simeq_D \rho(s')$;
- $\boldsymbol{d} \models s \in s'[\rho]$ if $\rho(s)$ and $\rho(s')$ are defined and $\rho(s) \in \rho(s')$;
- $\boldsymbol{d} \models D(x)[\rho]$ if $\rho(x) \in \pi^*(D)$;
- $\boldsymbol{d} \models \neg\phi[\rho]$ if $\boldsymbol{d} \not\models \phi[\rho]$; $\boldsymbol{d} \models \phi \wedge \psi[\rho]$ if $\boldsymbol{d} \models \phi[\rho]$ and $\boldsymbol{d} \models \psi[\rho]$; and $\boldsymbol{d} \models \phi \vee \psi[\rho]$ if $\boldsymbol{d} \models \phi[\rho]$ or $\boldsymbol{d} \models \psi[\rho]$;
- $\boldsymbol{d} \models \forall x \in D\, \phi\, [\rho]$ if for each $o \in \pi^*(D)$, $\boldsymbol{d} \models \phi\, [\rho(x/o)]$ and similarly, $\boldsymbol{d} \models \exists x \in D\, \phi\, [\rho]$ if there exists some $o \in \pi^*(D)$, $\boldsymbol{d} \models \phi\, [\rho(x/o)]$;
- $\boldsymbol{d} \models \forall x \in s\, \phi\, [\rho]$ if for each $v \in \rho(s)$, $\boldsymbol{d} \models \phi\, [\rho(x/v)]$ and similarly, $\boldsymbol{d} \models \exists x \in s\, \phi\, [\rho]$ if there exists some $v \in \rho(s)$, $\boldsymbol{d} \models \phi\, [\rho(x/v)]$.

Using the language we have just defined, integrity constraints of structural databases can be expressed.

**Definition:** Let $\boldsymbol{D} = (\boldsymbol{A}, \boldsymbol{C}, A, T, \simeq, \textbf{isa}, M, H)$ be a structural schema. An *integrity constraint over* $\boldsymbol{D}$ is a mapping $\Psi$ from $\boldsymbol{C}$ to rr-formulas (possibly null) such that for each $D \in \boldsymbol{A}$, the following condition is satisfied. Let $D_1, \ldots, D_n \in \boldsymbol{C}$ be all concrete domains for $D$, i.e., for each $i \in [1..n]$, $A(D_i) = D$, and $\phi_i = \Psi(D_i)$.

- $\phi_i$ is an rr-formula with only one free variable of the type $T_{D_i}$;
- For each structural instance $\boldsymbol{d} = (\pi, \delta)$ of $\boldsymbol{D}$, $v, v' \in \mathcal{V}_\pi(T_{D_i})$,
$$v \simeq_{D_i} v' \implies d \models \phi_i(v) \leftrightarrow \phi_i(v')$$
- For each $j \in [1..n], j \neq i$, each structural instance $\boldsymbol{d} = (\pi, \delta)$ of $\boldsymbol{D}$, and each $v \in \mathcal{V}_\pi(T_{D_i})$,
$$d \models \phi_i(v) \leftrightarrow \phi_j(H_{D_i \to D_j}(v))$$

Let $\boldsymbol{d} = (\pi, \delta)$ be a database instance of $\boldsymbol{D}$. $\boldsymbol{d}$ satisfies the constraint $\Phi$ if for each abstract domain $D \in \boldsymbol{A}$, the following is true:

- Let $D_1, \ldots, D_n$ be all concrete domains of $D$. Then, for each $i \in [1..n]$ and each $o \in \pi(D)$, $\boldsymbol{d} \models \Phi(D_i)(\delta(D_i, o))$.

Note that in a constraint rr-formula, all quantified variables are bounded. Given a structural database instance, the time to check if it satisfies all constraints depends on formulas. In particular, the set-nesting levels of the terms in the formulas will place time bounds. Most practical constraints typically do not have nesting levels $> 1$. In these cases, it is not hard (polynomial time) to check if all constraints are satisfied.

This model can be easily extended to include derived domains defined based on constraints. Examples of such derived domains include: *ConvexPolygons*, which are *Polygons* representing convex areas; *DEMs*, which are spatial mappings defined on every "grid" within the rectangular area, etc. When a schema contains such derived domains, in an instance all objects in domain *ConvexPolygons* (or *DEMs*) that satisfy that constraints must also be in the derived domain.

**Example 3.6:** Suppose a concrete domain *DEM-PtoR (PtoR)* of *DEMs* uses a type

$$\tau_{DEM} = [\mathit{Points}, \mathit{Points}, \mathit{Reals}, \mathit{Reals}, \mathit{Integers}, \mathit{Integers}, \mathit{Points} \rightarrow \mathit{Reals}]$$

to represent (rectangular) DEMs, where the two points are the lower left and upper right points of the boundary rectangle of the DEM, the two real numbers are the increments of the two dimensions, the two integers stand for the numbers of columns and rows, and the actual elevations are represented as a function. A constraint on *DEMs* states that the function is defined on every grid point within the rectangle. The constraint can be expressed as $\phi(x) = \psi(\nu^{PtoR}(x))$, where

$$\begin{aligned}\psi(y) = {} & \forall p_x \in [1..(y.5)]\, \forall p_y \in [1..(y.6)] \\ & \quad (\nu^{CP}(y.1).1 + (p_x - 1)y.3 \leq \nu^{CP}(y.2).1 \wedge \\ & \quad\ \nu^{CP}(y.1).2 + (p_y - 1)y.4 \leq \nu^{CP}(y.2).2 \\ & \quad \longrightarrow \\ & \quad (\exists p \in rg(y.7) \\ & \qquad\quad \nu^{CP}(p) = [(\nu^{CP}(y.1).1 + (p_x - 1)y.3), (\nu^{CP}(y.1).2 + (p_y - 1)y.4)]\end{aligned}$$

Note that since $y.1, y.2, p$ are points, a value function $\nu^{CP}$ is used to access the cartesian representation of the points, and *rg* is the function returns the range of a finite function. □

# 4 Spatio-Temporal Domains

We now exemplify the formalism presented above in terms of a set of abstract and concrete R-domains that involve spatio-temporal entities. Such domains have a fundamental and increasing importance for large numbers of applications, and particularly in the area of scientific databases. This set of R-domains is intended to be exemplary rather than exhaustive, and is aimed at demonstrating the power and flexibility of the basic model. We provide examples of both abstract and concrete R-domains and apply the two basic methods for creating new R-domains, and hence for constructing a large lattice of R-domains. These methods, as noted above, are the *application of constructors* to pre-existing domain elements and the *application of value constraints* to elements of pre-existing domains.

Using these methods, we define two distinct sets of domains. The first set includes several fundamental spatio-temporal domains. These domains should typically be available as part of a core set of domains for many classes of applications that involve the construction of models of complex spatio-temporal entities. Figure 5 illustrates schematically how a lattice of some of these domains may be organized. We note that while users might typically create a large number of new R-domains that are tailored to their applications, we should expect the system to provide a *core sublattice* of key domains as well as providing support for the inductive construction of new domains and transformations. An important subset of such core domains should relate to the domains of various *standards*, such as those of the recently adopted Spatial Data Transfer Standard (SDTS) [19]. Although the standard does not employ the explicit terminology of R-domains, it is easy to interpret the SDTS in terms of two classes of isomorphic R-domains that correspond to what we term an abstract R-domain. The first class of concrete R-domain is similar to many of the domains that we define below, while the second class provides specific data structures for the electronic transmission of instances of domain elements.

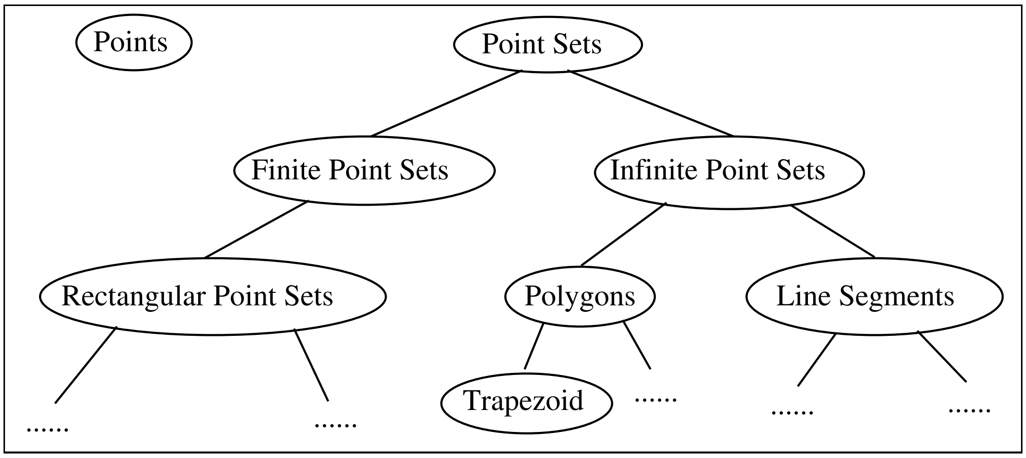


Figure 5: Spatial Domains

The second set of domains that we describe are examples of those that might be constructed for a specific application, and were introduced from that point of view in Section ??? We present informal definitions of the various domains that are presented implicitly in Figures ???

## 4.1 Spatio-temporal Domains and Spatial Projection Transformations

Spatio-temporal domains are fundamental for a large variety of applications and have been examined by a number of researchers. Some of the domains that we exemplify in this first set of examples have been discussed in [25] and are similar to some of the *realms* defined by Guting [15, 16]. In particular, the set of domains includes examples from four important classes of domains, which we term *pure spatial domains* (PSD), *pure temporal domains* (PTD), *spatio-temporal domains* (STD), and *geographic domains* (GD). In principal, PSD includes many of the domains that are defined in classical geometry. Our examples include classes of zero-, one- and two-dimensional primitives as well as some basic spatio-temporal and functional domains.

A fundamental and defining characteristic of classes of domains such as PSD, STD and GD is that any of their elements may be characterized in terms of an *essential and non-trivial projection onto some element of a domain in PSD.* For each domain, this projection is represented in terms of a special transformation associated with the domain, which we term the *spatial projection transformation.* Similarly each element of any domain in PTD and STD is characterized in terms of an essential and non-trivial projection onto some element of a domain in PTD and for each domain, this projection is represented in terms of a special transformation associated with the domain, which we term the *temporal projection transformation.* The significance of these transformations, and particularly of the spatial projection, arises from the fact that many expressions and queries involving elements of these four categories of domains directly involve these projections.

While it is possible to provide many different definitions of the spatial projection transformation, we have found it useful to define a *default* spatial projection $\prod_{\text{spatial}}$ which appears to arise naturally in many applications. This default transformation may be overridden with the definition of other spatial projections. In order to define the spatial projection transformation, we assume that all domains in PSD can be defined in terms of a set of superdomains that includes the R-domains

*Points* and *Pointsets* and that typically possess a large number of subdomains. Then

1. for an element $e$ of any concrete R-domains in PSD that satisfy the relation **isa** with respect to either *Points* or *Pointsets*:
$$\prod\nolimits_{\text{spatial}}(e) = e.$$
For example, the spatial projection of an element of *Linesegments* is defined to be the pointset comprising the linesegment and the spatial projection of an element of *Polygons* is defined to be the pointset comprising the polygon. These definitions are independent of the concrete representation of domain elements, which in the case of a linesegment may be the two endpoints and in the case of a Polygon may be a sequence of *Points* or *Linesegments*.

2. For other R-domains in PSD that do not satisfy the relation **isa** with respect to either *Points* or *Pointsets*, it must be the case that their representation involves in some essential manner the use some set of constructors $C_1, ..., C_n$ applied to elements of domains whose superdomains include either *Points* or *Pointsets*. The spatial projection of an element $e$ of such a domain is defined by distributing $\prod_{\text{spatial}}$ over the constructors in a recursive manner. For example, [3]
$$\prod\nolimits_{\text{spatial}}(C(e_1, \ldots, e_n)) = C(\prod\nolimits_{\text{spatial}}(e_1), ..., \prod\nolimits_{\text{spatial}}(e_n)).$$

3. For R-domains in STD or GD, the spatial projection of an element is defined by applying $\prod_{\text{spatial}}$ recursively to the representation of an element $e$ and removing any subelements for which $\prod_{\text{spatial}}(e)$ is not defined. For example, if $t_i,\ i = 1,\ n$ is an element of some domain in PTD and $p_i,\ i = 1,\ n$ satisfies **isa** with respect to *Points*, then
$$\prod\nolimits_{\text{spatial}}(\{[p_1,\ t_1], ...,\ [p_n,\ t_n]\}) = \{p_1, ...\ p_n\}$$

We may define the temporal projection transformation in similar terms.

Another important characteristic of many spatio-temporal domains is that their elements are frequently characterized in terms of an infinite number of subentities. *Polygons*, for example are characterized by an infinite number of *Points*. An important issue in constructing concrete R-domains for such elements is to find appropriate *finite representations* for such elements. Since it is not possible to provide an explicit representation of all the subentities, it is necessary to provide a special transformation for such domains that decides whether some arbitrary subentity belongs or not to a given domain entity. In the case of *Polygons* for example, it is necessary to provide a transformation $\in$ that examines whether a given point is in a given polygon. Another issue concerning finitely representable domains involves the class of such domains that one may express in terms of available languages.

In the definitions of domains provided in the subsections below, we employ the naming convention *name1⋆name2* for the names of concrete domains, in which *name1* is the name of the abstract R-domain and *name2* is a name indicating the concrete representation employed. It is sometimes convenient to leave *name2* as null, in which case the concrete domain name is *name1⋆*. We do not indicate all the transformations that may be applied to these domains but give certain representative examples.

[3]While it is clear that $\prod_{\text{spatial}}(e) = e$ is true for all domains in PSD, our definition of $\prod_{\text{spatial}}$ emphasizes the distributivity of the transformation.

## 4.2 Examples of Domains from PSD, PTD, STD and GD

The domains of PSD, PTD, STD and GD for which we provide definitions this paper include:

*Points* and *Points⋆Polar*, *Points⋆Cartesian*
*Pointsets*
*FinitePointsets*
*Linesegments* and *Linesegments⋆PointPairs*
*LinesegmentsSets*
*Arcs* and *Arcs⋆Pointsequences*, *Arcs⋆LineSequences*
*SimpleArcs*
*Loops*
*SimpleLoops*
*LoopSets*
*Strands*
*SimpleStrands*
*SimpleAreas* and *SimpleAreas⋆LoopSets*
*SimpleRegions*
*TimePoints*
*TimeIntervals*
*TimeSpans*
*TemporalSimpleRegions*
*RegionGeographies*

Schematic representations of typical elements of some of these domains are shown in Figure 5. This set of domains provide a useful set of relatively primitive domains that may be employed in the construction of large numbers of other domains, particularly by the application of constraints. Since the definitions of these domains are repetitive, we present a small number of these definitions in the current section, and list the balance in Appendix ??? The structure of a small lattice of typical domains from PSD is shown in Figure ???

In the following examples of domains from PSD, we assume for simplicity that we are modeling entities in Euclidean 2-space. It is straightforward to extend them to spaces with other dimensions. The set of transformations for each domain is merely illustrative and in particular, we omit definitions of the spatial projection transformations since these have already been defined in general for domains of PSD.

| | |
|---|---|
| **abstract domain:** | *Points* |
| **concrete domains:** | $*$*Points⋆Cartesian* |
| | *Points⋆Polar* |
| **operations:** | *distance* : *Points* $\times$ *Points* $\rightarrow$ *Reals* |
| **isomorphisms:** | *pol_car* : *Points* $\star$ *Polar* $\rightarrow$ *Points* $\star$ *Cartesian* **as** |
| | $pol\_car(x, y) = [x \cos y, x \sin y]$ |
| | *car_pol* : *Points* $\star$ *Cartesian* $\rightarrow$ *Points* $\star$ *Polar* **as** |
| | $car\_pol(x, y) = \ldots$ |

As noted above, *Points* is one of the two fundamental domains of PSD. The asterisk in front of the concrete domain *Points⋆Cartesian* indicates that this domain is the *canonical domain* associated with the abstract domain *Points*. In general, each abstract domain is associated with precisely one *canonical concrete domain*, which is referenced by other concrete domains through the given isomorphisms.

| | |
|---|---|
| **concrete domain:** | *Points⋆Cartesian (CP)* |
| **implements:** | *Points* |
| **structure:** | $[\mathit{Reals}, \mathit{Reals}]$ |
| **object equality:** | $p_1 \simeq_{CP} p_2$ iff $\phi(\nu^{CP}(p_1), \nu^{CP}(p_2))$ where $\phi(x, y) = (x = y)$ |
| **operations:** | $\mathit{distance}(x, y) = \sqrt{(x.1 - y.1)^2 + (x.2 - y.2)^2}$ |

| | |
|---|---|
| **concrete domain:** | *Points⋆Polar (PP)* |
| **implements:** | *Points* |
| **structure:** | $[\mathit{Reals}, \mathit{Reals}]$ |
| **object equality:** | $p_1 \simeq_{PP} p_2$ iff $\phi(\nu^{PP}(p_1), \nu^{PP}(p_2))$ where $\phi(x, y) = (x.1 = y.1 \wedge \exists k \in \mathit{Integers},\ x.2 - y.2 = k\pi)$ |
| **operations:** | *distance* (Derived from *Points⋆Cartesian*) |

Clearly, the equality predicate is an equivalence relation. Since in general the equality will be defined using a formula, we will only provide the formula in terms of the values, or even omit the formula when the equality is defined directly in terms of the equality of the domains used (e.g. *CP*) in the remainder of the paper. Operations are required to be well-defined with respect to the defined equality predicate, in the sense that equal points give equal results under the operations.

The next domain is the second of the two fundamental R-domains of PSD.

| | |
|---|---|
| **abstract domain:** | *PointSets* |
| **concrete domains:** | Many |
| **operations:** | $\in$: *Points* $\times$ *Pointsets* $\rightarrow$ *Booleans* |

Any domain in which sets of points are represented is a subdomain of this domain. In particular, we have two general classes of subdomains, *FinitePointsets* and *InfinitePointsets*. A key attribute of any subdomain of *InfinitePointsets* is that any element of such a domain possess a *finite representation*. The domain of *FinitePointsets*, which may be viewed in terms of applying the set constructor to the domain *Points* or applying a finiteness constraint to *Pointsets* is defined by the first method as:

| | |
|---|---|
| **abstract domain:** | *FinitePointsets* |
| **concrete domains:** | *FinitePointSets⋆* |
| **inherits:** | *Pointsets* |

| | |
|---|---|
| **concrete domain:** | *FinitePointSets⋆* |
| **implements:** | *FinitePointsets* |
| **structure:** | $\{\mathit{Points}\}$ |
| **operations:** | $\in = \cdots$ |

We note that in this case, we have to chosen to omit *name2* in the name for the concrete domain. In this case, we have described the structure of the domain in terms of abstract domain names. In defining domains with highly complex structures, we note that the number of concrete representations will in general increase exponentially with the depth of nesting defined in the structure. We may construct many subdomains of *FinitePointsets* by imposing constraints on the geometry and topology of the constituent points.

We now model some fundamental "two-dimensional" domains. Figure 6 shows some examples of

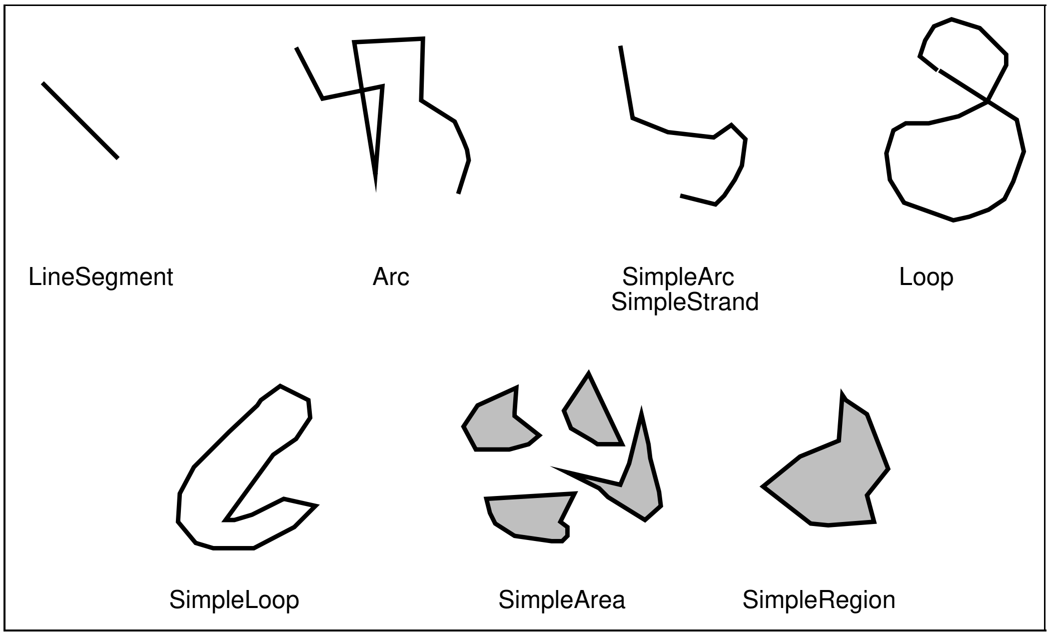


Figure 6: Purely Spatial Domains

the objects discussed.

**abstract domain:** *LineSegments (LS)*
**concrete domains:** *LineSegments⋆PointPair (LS⋆PP)*
**operations:** $\in$: *Points* $\times$ *LS* $\rightarrow$ *Boolean*
*length* : *LS* $\rightarrow$ *Reals*
*nodes* : *LS* $\rightarrow$ *PointSets*
*intersection* : *LS* $\times$ *LS* $\rightarrow$ *LS*

This domain is clearly a subdomain of *InfinitePointsets*. Since it is not possible to list all the members of such a set, transformations that define the elements of such a domain is required. The transformation $\in$ is intended to serve this purpose, and returns 1 if some arbitrary *Point* is contained in the *Linesegment* and 0 otherwise. In general, we must provide all members of the class of *InfinitePointsets* with such a transformation. The transformation *nodes* simply returns the two end-nodes of the segment; *intersection* returns the line segment, if it exists, formed from the intersection of the two line segments. This line segment may represent a point, as the degenerate case.

**concrete domain:** *LineSegments⋆PointPairs (PPLS)*
**implements:** *Linsegments*
**structure:** $[Points, Points]$
**object equality:** $[x, y] = [y, x]$
**operations:** $length(x, y) = distance(x, y)$
$nodes(x, y) = \{x, y\}$
*intersection* : . . .
$\in$ : . . .

The implementation of operations such as ‘intersection’ and (as with others below) is not given in full but is a routine application of coordinate geometry.

**abstract domain:** *LineSegmentSets*
**concrete domain:** *LineSegmentSets*$\star$
**structure:** $\{$*LineSegments*$\}$

**abstract domain:** *Arcs*
**concrete domains:** $*$*Arcs*$\star$*Pointsequences (PSA)*
*Arcs*$\star$*LineSequences (LSA)*
**operations:** *length* : *Arcs* $\rightarrow$ *Reals*
*extremes* : *Asrc* $\rightarrow$ *PointSets*
*intersection* : *Arcs* $\times$ *Arcs* $\rightarrow$ *LineSegmentSets*
*selfintersect* : *Arcs* $\rightarrow$ *Booleans*
**isomorphisms:** *psa_lsa* : *PSA* $\rightarrow$ *LSA* **as**
$$psa\_lsa(\langle x_1, \ldots, x_n \rangle) = \langle (x_1, x_2), \ldots, (x_{n-1}, x_n) \rangle$$

The transformations are explained in the following. *Extremes* returns the two extreme points (if they exist) of the arc. *Intersection* returns the set of lines and points (if they exist) comprising the intersection of the two arcs *Selfintersect* returns *true* if the arc intersects itself.

**concrete domain:** *Arcs*$\star$*Pointsequences (PSA)*
**implements:** *Arcs*
**structure:** $\langle$*Points*$\rangle$
**operations:** $length(\langle x_1, \ldots, x_n \rangle) = \sum_{i=1}^{n-1} distance(x_i, x_{i+1})$
$extremes(\langle x_1, \ldots, x_n \rangle) = \{x_1, x_n\}$
*intersection* (Derived from LSA)
*selfintersect* (Derived from LSA)

**concrete domain:** *Arcs*$\star$*LineSequences (LSA)*
**implements:** *Arcs*
**structure:** $\langle$*LineSegments*$\rangle$
**operations:** *length* (Derived from PSA)
*extremes* (Derived from PSA)
$intersection(\langle x_1, \ldots, x_m \rangle, \langle y_1, \ldots, y_n \rangle)$
$= \{intersection(x_i, y_j) \mid intersection(x_i, y_j) \neq \emptyset\}$
*selfintersect* : $\langle x_1, \ldots, x_n \rangle = true$ iff
$\exists i \exists j,\ i > j + 1$ and $intersection(x_i, x_j) \neq \emptyset$

We now consider some fundamental temporal and spatiotemporal domains. The domains given below are a small number from a large set of possibilities. Three purely temporal domains (PSD) include:

**abstract domain:** *TimePoints*
**concrete domain:** *TimePoints*$\star$
**structure:** *Reals*

**abstract domain:** *TimeIntervals*
**concrete domain:** *TimeIntervals*$\star$
**structure:** $[$*TimePoints*, *TimePoints*$]$
**operations:** (Included here would be all the operations discussed, for example, by Allen in [3])

**abstract domain:** *TimeSpans*
**concrete domain:** *TimeSpans*$\star$
**structure:** $\{$*TimeIntervals*$\}$
**constraints:** $\forall x \in$ *TimeSpans* $\forall y, z \in x.\ \ y \cap z = \emptyset$
(Each member of 'TimeSpans' is a set of disjoint time intervals.)

The purely temporal domains above can be combined with spatial domains to give mixed spatiotemporal domains, which are of great utility in many applications. We give a single example.

**abstract domain:** *TemporalSimpleRegions*
**concrete domain:** *TemporalSimpleRegions*$\star$
**structure:** *TimeSpans* $\rightarrow$ *SimpleRegions*

Each member of this domain is an association of a set of time intervals with a simple region, indicating the times in which the region is extant. The spatial projection of an element from this domain would clearly be the *SimpleRegions* that define the element.

Many geographical entities may be represented in terms of functions from spatiotemporal domains to domains of attributes. Simple examples of such domains include representations of landsurfaces in terms of a mapping from representations of position to representations of topographic altitude; levels of rainfall as a mapping from time and position on an irregular grid into a representation of rainfall rates. representations of regional population levels as mappings from representations of time pointsets into representations of population levels. We give one example of a primitive geographic domain of this type.

**abstract domain:** *RegionMappings*
**concrete domain:** *RegionalMappings*$\star$
**structure:** *TemporalSimpleRegions* $\rightarrow$ *Attributes*

Each member of this domain is a temporally labelled simple region, with which is associated an attribute value.

## 4.3 Examples of Some R-domains for a Hydrological Application

Finally we briefly outline a set of R-domains that exemplify those that might be constructed for a specific application. Some of these R-domains are employed in the examples in Sections 2, 3,5, and 6, where it is shown how some of their definitions may be expressed in MDB.

We provide in the following a description of the domains in the application. The description is mostly informal, although it is easy to express them using the data model.

1. *RainfallRates* and *Elevations*: are simple abstract R-domains for which typical concrete representations would include *RainfallRates*$\star$*PositiveReals* and *Elevations*$\star$*Reals* respectively.

2. An R-domain for digital elevation models *DEMs* with structure defined either by $\{[$*Points*, *Elevations*$]\}$ or as a mapping, together with the associated constraint that the *spatial projection* of any element in the domain be an element of the domain *FiniteLatticePointSets*. One may associate with such domains transformations such as *make-drainage-net* which takes and element of *DEMs* as input and produces an instance of a domain in which river networks are represented.

3. An R-domain of the slopes of landsurfaces, represented by *DEMs*, which we name *DEM-Slopes*.
4. The abstract domain *Rainfalls* contains both the measured interpolated and rainfall datasets. One may associate with such domains transformations such as *rainset-interpolate* and *find-storms*. The former transformation, for example, might produce a maximally-connected pointset over which the rainfall rate exceeds a constrained value during a limited time interval. A concrete domain of representations for the abstract R-domain *Rainfalls* may either be associated with structure: $\{[\mathit{Points}, \mathit{TimePoints}, \mathit{RainfallRates}]\}$ or represented as mapping.
5. An R-domain of *Storms* that are constructed by applying *find-storms* to instances of the domains of *Storms*.
6. An R-domain of *Hydrographs*, which may be represented in terms of a mapping from some representation of time points to some representation of discharge in a river.

# 5 A Manipulation Language

In this section we describe a database manipulation language for expressing queries and for defining transformations. The language combines features from languages of the complex objects or nested relations model [1] and of object-oriented models [4]. Specifically, the languages allows manipulations on objects, transformations, and complex structured values built from the type constructors provided in the model. Predicates from interpreted domains (e.g., *Integers*, *Boolean*, *Time*) are permitted. It can also include the object equality, used-defined functions and predicates (transformations). Specifically, the design of the language is based on the constraint language defined in Section 3.3. The notions of "queries" and "function expressions" are formally presented. Queries generally returns structured values as answers, similar to the conventional notion of queries. Functon expressions are used to define single-valued or multivalued functions or transformations in a simple manner. Queries and function expressions may then be used as primary components in a simple modeling and database language MDBL that is described in the following section. We begin with an informal discussion of the notion of "query", illustrated by several examples, and then introduce the manipulation language, queries, and function expressions.

Informally, we define queries as expressions of form $\{(x_1, \ldots, x_n) \mid \phi(x_1, \ldots, x_n)\}$, where $x_1, \ldots, x_n$ are variables and $\phi$ is a formula with only variables $x_1, \ldots, x_n$ occurring free in $\phi$. Using rr-formulas as query formulas, we can write some simple queries.

The semantics of the answer is defined in the usual manner. Let $\boldsymbol{d} = (\pi, \delta)$ be a database structural instance. The *active domain* of a type $\tau$, denoted by $\mathit{adom}_\tau(\boldsymbol{d}) \in \mathcal{B}$, is defined as follows. If $\tau \in \mathcal{B}$, $\mathit{adom}_\tau(\boldsymbol{d}) = \mathcal{V}(\tau)$. If $\tau \in \boldsymbol{A}$, $\mathit{adom}_\tau(\boldsymbol{d})$ is the set of all oid's occurring in $\boldsymbol{d}$, i.e. $\mathit{adom}_\tau(\boldsymbol{d}) = \cup\{\pi(D) \mid D \in \boldsymbol{A}\}$. Finally, $\mathit{adom}_\tau(\boldsymbol{d})$ is obtained naturally if $\tau$ is defined using type constructors.

The *answer* to the query $\{(x_1, \ldots, x_n) \mid \phi(x_1, \ldots, x_n)\}$, where $x_i$ is of type $\tau_i$, on an input database $\boldsymbol{d}$ is the set:

$$\{(e_1, \ldots, e_n) \mid \boldsymbol{d} \models \phi[(x_1/e_1) \cdots (x_n/e_n)], \forall i \in [1..n], e_i \in \mathit{adom}_{\tau_i}(\boldsymbol{d})\}$$

**Example 5.1:** Suppose *RainFalls (RF)* is an abstract domain and *RF-PxR* is a concrete domain with structure $\{[\mathit{Points}, \mathit{Reals}, \mathit{Time}]\}$. We further assume that $t_0$ and $t_{100}$ are two time instants. The following query retrieves all rainfall measures ($\geq 0.5$ inches) within the specified time frame $[t_0, t_{100}]$.

$$\{(x) \mid \exists o_r \in \mathit{RF}\ x \in \nu^{\mathit{RF\text{-}PxR}}(o_r) \wedge x.2 \geq 0.5 \wedge t_0 \leq x.3 \leq t_{100}\}$$

□

**Example 5.2:** Recall that each storm can be represented by a set of tuples specifying the point, time and amount of rain measured. Suppose *area* : *Storm* → *Reals* is a transformation signature. The following query selects the storm which covers the largest area.

$$\{x \mid x \in \mathit{Storm} \wedge \forall y \in \mathit{Storm}\ (\mathit{area}(y) \leq \mathit{area}(x))\}$$

□

In practice, it is convenient to allow "nested queries". We can incorporate such nested queries by introducing "query terms". Briefly, a *query term* is a term of type $\{[\tau_1, \ldots, \tau_n]\}$ that may be expressed in the form $\{(x_1, \ldots, x_n) \mid \phi(x_1, \ldots, x_n)\}$, where variables $x_1, \ldots, x_n$ occur free in $\phi$ ($\phi$ may have other free variables). Let $\rho$ be an assignment. We also generalize $\rho$ naturally to include query terms (with respect to a databse $\boldsymbol{d}$):

$$\rho(\{(x_1, \ldots, x_n) \mid \phi(x_1, \ldots, x_n)\}) = \{(e_1, \ldots, e_n) \mid \boldsymbol{d} \models \phi[\rho(x_1/e_1) \cdots (x_n/e_n)]\}.$$

**Example 5.3:** We consider again Example 5.2. Now suppose the transformation *area* is define as of the type $\{\mathit{Points}\} \rightarrow \mathit{Real}$, i.e., the specific *area* function for the domain *Storm* is not available. Now the same query can be expressed as a nested query:

$$\{x \mid x \in \mathit{Storm} \wedge \forall y \in \mathit{Storm}\ (\mathit{area}(\{p|\phi(p, y)\}) \leq \mathit{area}(\{p|\phi(p, x)\}))\}$$

where $\phi(p, z) = \exists t \exists r [p, r, t] \in \nu^{\mathit{RF\text{-}PxR}}(z)$. □

The rr-formulas are very restrictive. Complicated queries are not expressible by them. We now present an extension of the basic language which allows more complicated queries to be expressed. Intuitively, we add "unrestricted" existential quantifiers to rr-formulas used in queries. Intuitively, a user may assert the existence of elements that do not already exist in the database. This greatly increases the expressive power of the language to the degree that it is capable of expressing all computable "generic" mappings on structural database instances. On the other hand, in databases that allow the set construct, the combination of negation and such existential quantifiers may increase the expressive power of the query language beyond computability [10]. Query languages that allow noncomputable queries are not realistic since they are not implementable. Hence in our language, we do not allow negation over such existential quantifiers. This restriction limits the expressive power of our language to only computable queries.

We begin with the following definitions of "query formulas", "queries", and "answers" to queries. We illustrate these notions with the use of examples. Intuitively, query formulas are rr-formulas extended with normal (non-range restricted) existential quantifiers. query formulas can also be constructed from other query formulas by conjunction, disjunction, and range restricted quantifications.

**Definition:** Let $\boldsymbol{D}$ be a database structural schema. The family of *query formulas* are defined as follows:

1. an rr-formula is a query formula;
2. $\exists x\, \phi$ is a query formula, if $\phi$ is a query formula and $x$ is a variable of type $\tau$;
3. $\phi \wedge \psi$ and $\phi \vee \psi$ are query formulas if $\phi, \psi$ are query formulas;
4. $\forall x \in s\, \phi$ and $\exists x \in s\, \phi$ are query formulas if $\phi$ is a query formula, $x$ is a variable of type $\tau$ and $s$ a term of type $\{\tau\}$.

**Example 5.4:** Consider a line segment represented by two end points, where points are represented by a pair of real numbers. Let $l$ be a line segment and further suppose the formula $\phi_{on}(p, l)$ states that the point $p$ is on the line segment $[p_1, p_2]$: Now, give two line segments, the following formula $\phi_{int}(l_1, l_2)$ expresses the intersection of them:

$$\phi_{int}(l_1, l_2) = \exists p\, \phi_{on}(p, l_1) \wedge \phi_{on}(p, l_2)$$

Obviously, $\phi_{int}$ is a query formula. □

**Definition:** A *query* over a database schema $\boldsymbol{D}$ is an expression of the form $\{(x_1, \ldots, x_n) \mid \phi(x_1, \ldots, x_n)\}$ where $x_1, \ldots, x_n$ are variables and $\phi(x_1, \ldots, x_n)$ is a query formula with only free variables in $\{x_1, \ldots, x_n\}$.

To define the semantics of queries, we need to extend the "satisfaction" by a database instance to query formulas.

**Definition:** Let $\phi$ be a query formula over $\boldsymbol{D}$ and $\rho$ a variable assignment. Further suppose $\boldsymbol{d}$ is a database instance. $\boldsymbol{d}$ *satisfies* $\phi$ *under* $\rho$, denoted as $\boldsymbol{d} \overset{q}{\models} \phi[\rho]$, is defined recursively as follows.

- $\boldsymbol{d} \overset{q}{\models} \phi[\rho]$ if $\phi$ is an rr-formula and $\boldsymbol{d} \models \phi[\rho]$;
- $\boldsymbol{d} \overset{q}{\models} \exists x\, \phi[\rho]$ if there exists some element $e \in adom_\tau(\boldsymbol{d})$, $\boldsymbol{d} \overset{q}{\models} \phi[\rho(x/e)]$
- $\boldsymbol{d} \overset{q}{\models} \phi \wedge \psi[\rho]$ if $\boldsymbol{d} \overset{q}{\models} \phi[\rho]$ and $\boldsymbol{d} \overset{q}{\models} \psi[\rho]$, and similarly $\boldsymbol{d} \overset{q}{\models} \phi \vee \psi[\rho]$ if $\boldsymbol{d} \overset{q}{\models} \phi[\rho]$ or $\boldsymbol{d} \overset{q}{\models} \psi[\rho]$;
- $\boldsymbol{d} \overset{q}{\models} \forall x \in s\, \phi[\rho]$ if for each element $e \in \rho(s)$, $\boldsymbol{d} \overset{q}{\models} \phi[\rho(x/e)]$ and similarly, $\boldsymbol{d} \overset{q}{\models} \exists x \in s\, \phi[\rho]$ if there exists some element $e \in \rho(s)$, $\boldsymbol{d} \overset{q}{\models} \phi[\rho(x/e)]$.

Since query formulas are an extension to rr-formulas, we also denote $\overset{q}{\models}$ simply as $\models$.

**Definition:** Given a query $Q = \{(x_1, \ldots, x_n) \mid \phi(x_1, \ldots, x_n)\}$ over database schema $\boldsymbol{D}$ and a database instance $\boldsymbol{d} \in inst(\boldsymbol{D})$, the *answer to* $Q$ *under* $\boldsymbol{d}$, denoted as $Q(\boldsymbol{d})$, is defined as the set (suppose $x_i$ is of type $\tau_i$):

$$Q(\boldsymbol{d}) = \{(e_1, \ldots, e_n) \mid \boldsymbol{d} \models \phi[(x_1/e_1) \cdots (x_n/e_n)], \forall i \in [1..n], e_i \in adom_{\tau_i}(\boldsymbol{d})\}$$

As it was stated in the informal discussion, we allow queries to be in nested forms, namely, a query formula can also contain terms which are queries.

In the formal setting, we extend the definition of *term* to include expressions of the following form, called *query terms*:

$$\{(y_1, \ldots, y_m) \mid \bigwedge_{1 \le i \le m} y_i \in s_i \wedge \phi(x_1, \ldots, x_n)\}$$

where $\phi$ is a query formula with free variables $x_1, \ldots, x_n$, $s_i$'s are terms (of appropriate types), and $\{y_1, \ldots, y_m\} \subseteq \{x_1, \ldots, x_n\}$. Let $\boldsymbol{d}$ be a database instance. For each variable assignment $\rho$, $\rho$ is extended to include query terms $s$ of the above form in the natural fashion (with respect to $\boldsymbol{d}$):

$$\rho(s) = \{[e_1, \ldots, e_m] \mid \forall i \in [1..m]\; e_i \in \rho(s_i), \boldsymbol{d} \models \phi[\rho(y_1/e_1) \cdots (y_m/e_m)]\}.$$

The following example illustrates the use of such nested queries.

**Example 5.5:** Suppose we have a rainfall dataset $r$ which is of type $\{[\mathit{Points}, \mathit{Reals}, \mathit{Time}]\}$ and a DEM $e$ of type *Points* $\rightarrow$ *Reals*, where *Time* is a discrete domain of time instants. Consider the operation to interpolate the measured rainfall over all points that have elevation provided in $e$. Suppose $p_1, \ldots, p_n$ are all points that have rainfall measures at the time instant $t$. The interpolation uses the following formula: for each point $p$:

$$\mathit{rainfall}(p) = \frac{\sum_{i=1}^{n} \mathit{rainfall}(p_i) \cdot \mathit{distance}^{-2}(p, p_i)}{\sum_{i=1}^{n} \mathit{distance}^{-2}(p, p_i)}$$

The operation can be expressed in the following steps.

First, we construct a 5-ary relation $y$ of type $\{[\mathit{Points}, \mathit{Points}, \mathit{Time}, \mathit{Reals}, \mathit{Reals}]\}$ in which each tuple represents a point $p$ (first column) from $e$ and another $p_i$ (second) from $r$, the time instant $t$, $\mathit{distance}^{-2}(p, p_i)$, and the rainfall at $p_i$ over $\mathit{distance}^{-2}(p, p_i)$. The construction is performed by the following formula:

$$\phi_1(y) = \forall x_r \in r\, \forall x_e \in \mathit{dm}(e)\, \exists x_y \in y\; \phi_2 \wedge \forall x_y \in y\, \exists x_r \in r\, \exists x_e \in \mathit{dm}(e)\; \phi_2$$

where

$$\phi_2 = \left(x_y = \left[x_e, x_r.1, x_r.3, \mathit{distance}^{-2}(x_e, x_r.1), \frac{x_r.2}{\mathit{distance}^{-2}(x_e, x_r.1)}\right]\right)$$

We then construct the output that is a relation of type $\{[\mathit{Points}, \mathit{Reals}, \mathit{Time}]\}$ using $\phi_1$:

$$\begin{aligned} Q = \{(z_p, z_r, z_t) \mid \exists y\; \phi_1(y) \wedge \exists v_d\, \exists v_n\; z_r = v_n / v_d \wedge \\ v_d = \mathrm{sum}\{u \mid \exists x_y \in y\; \phi_3 \wedge u = x_y.4\} \wedge \\ v_n = \mathrm{sum}\{u \mid \exists x_y \in y\; \phi_3 \wedge u = x_y.5\}\} \end{aligned}$$

where $\phi_3 = z_p = x_y.1 \wedge z_t = x_y.3$. $\square$

We can view database queries as mapping from *inst*$(\boldsymbol{D})$ to *inst*$(\tau)$ (for some type $\tau \in \boldsymbol{\mathcal{T}}$) which are "generic" [5, 8]. The notion of genericity means that the language will treat data elements generically or independent of how they will be stored. Namely, every automorphism on an input database instance (respect functions) should also be an automorphism on the output. We can then define the class of computable queries to be the set of all generic mappings that are computable. Using results from [10, 2], it can be verified that the language defined here expresses exactly all computable queries (modulo copies).

In the remainder of the section, we show that the query language can also be used to define transformations. One subtlety is that a query generally returns a set of elements on an input database, i.e., they define multi-valued functions. In our language, however, we allow query formulas to define both single- and multi-valued functions and the result may not be defined if more than one elements are in the query answer.

We define a *single-valued function expression* to be of the form

$$t(x_1, \ldots, x_n) = y \ \text{ where } \ \phi(x_1, \ldots, x_n, y)$$

and a *multi-valued function expression* to be of form:

$$t(x_1, \ldots, x_n) = \{y \mid \phi(x_1, \ldots, x_n, y)\}$$

where $\phi$ is a formula with free variables $x_1, \ldots, x_n, y$. Let $\tau_1, \ldots, \tau_n, \tau$ be the types of $x_1, \ldots, x_n, y$ (resp.). The above function expression then defines a function of type $(\tau_1, \ldots, \tau_n) \rightarrow \tau$ (single-valued) or $(\tau_1, \ldots, \tau_n) \rightarrow \{\tau\}$ (multi-valued).

Given a database $\boldsymbol{d}$, let $Q(\boldsymbol{d})$ be the answer

$$Q(\boldsymbol{d}, e_1, \ldots, e_n) = \{e \mid \boldsymbol{d} \models \phi[(x_1/e_1) \cdots (x_n/e_n)(y/e)]\}$$

with respect to $\rho$, where $rho(x_i) = e_i$ for each $i \in [1..n]$. Then the multi-valued function express above defines the function $t_m$: $t_m(e_1, \ldots, e_n) = Q(\boldsymbol{d}, e_1, \ldots, e_n)$. Finally, the single-valued function expression defines a function $t_s$:

$$t_s(e_1, \ldots, e_n) = \begin{cases} e & \text{if } \ Q(\boldsymbol{d}, e_1, \ldots, e_n) = \{e\} \ \text{ is a singleton set} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Note that in general the functions defined using the expressions of the above forms are relative to the database $\boldsymbol{d}$.

**Example 5.6:** As an example, we consider a slightly more complicated query. Suppose one concrete domain of *DEMs*, *DEM-PtoR*, uses a function from points to elevation. From a DEM, one may compute a surface *slope*, which may be viewed as a spatial mapping from points to directions that indicates the flow directions of surface water. We can use an integer range type [1..4] to represent the direction, and a binary relation of type $\{[\textit{Points}, [1..4]]\}$ to represent a slope.

Now suppose neighbor relations *east*, *south*, *west*, *north*, *neighbor* are defined. We can use the language to define the transformation *dem-slope*$(m : \textit{Points} \rightarrow \textit{Reals}) : \{[\textit{Points}, [1..4]]\}$

$$\begin{aligned} \textit{direction}(p, q) = x \ \text{ where } \ & x = 1 \wedge \textit{east}(p, q) \vee x = 2 \wedge \textit{north}(p, q) \vee \\ & x = 3 \wedge \textit{west}(p, q) \vee x = 4 \wedge \textit{south}(p, q) \end{aligned}$$

$$\textit{dem-slope}(m) = \{(p, d) \mid \phi(p, d, m)\}$$

where

$$\phi(p, d, m) = p \in \textbf{\textit{dm}}(m) \wedge \exists q(m(q) = \min\{m(q') \mid \textbf{\textit{neighbor}}(p, q')\} \wedge d = \textbf{\textit{direction}}(p, q))$$

□

Datasets in the real world are represented as objects in our data model and each object has a unique identifier. In the application programs and transformations, it is necessary to create new objects, which include generating new object identifiers and "attach" structured valued to them. To include such a functionality, we introduce an operation for creating object identifiers. We first illustrate with an example.

**Example 5.7:** We consider again Example 5.6, where for a given input *DEM*, a slope is generated. We also assume that there is an abstract domain *DEM* for representing all DEMs. A concrete domain *DEM-PtoR* of *DEM* uses a structure *Points* → *Reals*. Similarly, slopes are also represented by an abstract domain *Slope* which has a concrete domain with a structure [*Points*, *Reals*]. Now the transformation *dem-slope* : *DEM* → *Slope* can be expressed as:

$$\textit{dem-slope}(o_m) = \textit{new}\ o_s \ \text{ where }\ \nu(o_s) = \{(p, d) \mid \phi(p, d, \nu(o_m))\}$$

where *direction* is defined in the same way and the formula $\phi$ is the same as in Example 5.6. The *new* operator in the above expression generates a new object identifier which does not exist in the current database. □

In general, we allow two ways in which transformations may create new oids: they may generate a single object or they may generate a set of objects. Formally, an *object creating* function expression is of either of the following forms:

$$t(x_1, \ldots, x_n) = \textit{new}\ y \ \text{ where }\ \phi(\nu(y), x_1, \ldots, x_n)$$
$$t(x_1, \ldots, x_n) = \{\textit{new}\ y \mid \phi(\nu(y), x_1, \ldots, x_n)\}$$

The semantics for a single-valued function is straightforward: a new oid is produced and its value is the return value of the same type as $\nu(y)$ and the function is undefined if there is more than one return value. The semantics for a multi-valued function is defined similarly: a new oid is created for each distinct value for $y$.

We end the section with the follwoing example.

**Example 5.8:** Consider the concrete domain *RF-PxR* for the domain *RainFall*. The type for *RF-PxR* is a set of pairs, i.e., $\{$[*Points*, *Reals*]$\}$. For simplification, we omit the associated time instants. We further suppose that the rainfall datasets have been transformed by interpolation, over all points in a DEM, for example. Recall that a storm, which is an element in the domain *Storms*, is a connected area in which each point has a measured rainfall greater than some threshold value (e.g., $\geq$ 0.5 inches). It is then natural to express the computation that on input an (interpolated) rainfall dataset produces all storms within the area. The transformation, *find-storms*, is of type *RainFall* → *Storm*. Let *S-PxR* be a concrete domain of *Storm* with type $\{$[*Points*, *Reals*]$\}$. Then *find-storms* on *S-PxR* can be expressed in the following steps. Let $r$ be an input rainfall object. First we define

$$r_{\text{n}} = \{(x_p, x_r) \mid [x_p, x_r] \in \nu^{\textit{RF-PxR}}(r) \wedge x_r \geq 0.5\}$$

Then we can express:

$$\textit{find-storms}(r) = \{\textit{new } x \mid \phi(\nu^{\textit{S-PxR}}(x), r_{\text{n}})\}$$

where $\phi$ is the following formula:

$$\begin{aligned}\phi(x_{\text{storm}}, y) = \exists x_{\text{set}} \in \textit{pset}(y) \ & \phi_{\text{StormSet}}(x_{\text{set}}) \wedge \\ & \forall z_{\text{set}} \in \textit{pset}(y) \ \phi_{\text{StormSet}}(z_{\text{set}}) \rightarrow \text{count}(x_{\text{set}}) \geq \text{count}(z_{\text{set}}) \wedge \\ & x_{\text{storm}} \in x_{\text{set}}\end{aligned}$$

and $\phi_{\text{StormSet}}(u) = \phi_{\text{disj}}(u) \wedge \phi_{\text{nei}}(u)$. Intuitively, $u$ is a set of sets of point rainfall pairs. The formula $\phi_{\text{disj}}$ states that sets in $u$ are pairwise disjoint, and the formula $\phi_{\text{nei}}$ asserts that for each point in $y$, if the point is also in a set $s$ in $u$, then all its neighbors that are in $y$ are also in the same set $s$.

The condition "$\forall z_{\text{set}} \in \textit{pset}(y) \ \phi_{\text{StormSet}}(z_{\text{set}}) \rightarrow \text{count}(x_{\text{set}}) \geq \text{count}(z_{\text{set}})$" is to ensure that points in disconnected areas are in different storms in $x_{\text{set}}$. □

# 6 MDBL: A High-level Modeling and Database Language

Based on our conceptual data model of domains and transformations, we have designed a simple, declarative, high-level *M*odeling and *D*ata*b*ase *L*anguage, *MDBL*, to allow investigators to express easily and naturally most of the operations that are employed in the iterative development of their models, while hiding computational issues that are irrelevant to the scientific goals of the investigators. It is designed to provide support for about 80-90% of scientific activities in terms of a very simple functionality. As noted in our characterization of scientific activity, the most important activities in scientific investigations may be characterized as:

1. defining new domains, new domain elements, new transformations and storing them in some database;
2. searching for datasets that satisfy certain constraints;
3. searching for transformations that satisfy certain constraints;
4. applying transformations domain elements in general and to datasets in particular.

Hence MDBL has been designed to provide scientists with straightforward support for creating, retrieving, storing, and deleting domains, transformations, and elements. Queries and function expressions defined in the previous section are used for specifying *what* a scientist wishes to access in a database and *how* computations should be carried out.

As one may note from Figure 7, the language supports four sets of high-level commands that are intended to support the four major classes of scientific activity. We briefly describe these commands and the provide examples to illustrate the use of the language. The first set of commands (`access`) permits accessing of information on domains, transformations and domain

$$\left\{\begin{array}{l}\texttt{ACCESS}\\ \texttt{CREATE}\\ \texttt{STORE}\\ \texttt{DELETE}\end{array}\right\}\left\{\begin{array}{l}\texttt{DOMAIN name}\\ \texttt{TRANSFORMATION name}\\ \texttt{ELEMENT name IN domain-name}\end{array}\right\}\texttt{WHERE conditions}$$

Figure 7: MDBL Commands Overview

elements from public and private databases. The second set of commands (`create`) permits the creation of new domains, new transformations and new domain elements. In particular new transformations may be created by a variety of methods, including function expressions that involve existing transformations, while new elements may be constructed by applying transformations to existing or constructed elements. The third set of commands (`store`) permits the storage of newly constructed domains, transformations and elements in the public database. The final set of commands (`delete`) permits deletion of domains, transformations and elements. The `where` conditions in each of the commands are specified declaratively in terms of a simple syntax. MDBL combines the conventional ideas of "data definition" and "data manipulation" languages.

MDBL permits system access at a variety of levels appropriate for different classes of users and tasks, and is designed to facilitate the translation of its commands in terms of an easy-to-use visual language that may be used as the basis for constructing a graphical interface. A user commences an MDBL session by specifying a "public" database (shared by many users) and a "private", persistent workspace. The workspace provides a database and computational modeling environment for scientists, and in particular permits the creation of new domains, elements, transformations as well as persistent variables that can hold objects (identifiers) and structured values. Both the public database and the private database may be accessed, and domains, transformations, and elements may be moved to the public database. A version of MDBL is currently being implemented, and implementation issue and future plans will be discussed in Section 7.

To exemplify the simplicity and the use of MDBL, we present a few simple examples. More complete examples are provided in the following section.

**Example 6.1:** The abstract domain of *Points* can be created using the command `create`:

`create abstract domain` *Points*
`where transformation` $\mathit{distance} : (\mathit{Points}, \mathit{Points}) \rightarrow \mathit{Reals}$

`create concrete domain` *Cartesian* `implements` *Points*
`type =` $[\mathit{Reals}, \mathit{Reals}]$
`transformation` $\mathit{distance}(x, y) = \mathrm{sqrt}((x.1 - y.1)^2 + (x.2 - y.2)^2)$

`create concrete domain` *Polar* `implements` *Points*
`type =` $[\mathit{Reals}, \mathit{Reals}]$
`transformation` $\mathit{distance}(x, y) = \mathrm{sqrt}((x.1 - y.1)^2 + (x.2 - y.2)^2)$
`object equality:` $x = y$ `iff` $x.1 = y.1$ `and` $\mathrm{E}\,k.(x.2 - y.2 = 2 * pi * k)$
`transformation` *distance* `is derived`

Since the domain *Points* has two concrete domains, isomorphisms between them need to be defined:

`create isomorphism` *pol2car* `from` *Polar* `to` *Cartesian* `in` *Points*

$\quad pol2car(x) = [x.1 \cos x.2, x.1 \sin x.2]$

We then use the `store` command to store the domain *Points* in the public database:

`store` *Points* □

**Example 6.2:** Consider again the rainfall datasets. Suppose a concrete domain *RF-Tuples* of the domain *RainFall* uses the type $\{[Points, Reals, Year, Month, Day, Hour]\}$ to represent hourly measured rainfall at various locations with rain gauges. We assume that a transformation *interpolation* is defined on *RainFall*, which is of type $[RainFall, DEM] \rightarrow RainFall$.

We first create a new rainfall object which only contains rainfalls for January 21, 1989 between hours 1 to 12:

`create element` $Y$ `in` *RF-Tuples*

`where value=`$\{x \mid \exists r \in RF\text{-}Tuples\,.\,(x \in \texttt{value}(r),\ x.Year = 1989,$
$\quad x.Month = 1,\ x.Day = 21,\ 1 \leq x.Hour \leq 12)\}$

The above command creates an element in *RainFall* and at the same time, sets the value of $Y$ to be the object identifier of the new object. Let $X$ be another variable holding an object identifier of a DEM. We may then interpolate the rainfall $Y$ with respect to $X$:

`create element` $Z$ `in` *RF-Tuples*

`where` $Z =$ `apply` *RainFall.interpolate* `to` $(Y, X)$ □

Variables can also be defined by using `access` command. The following example retrieves a set of DEMs and stored them in a variable $W$:

**Example 6.3:** Suppose *Manaus* is an object representing the area of Manaus in Amazon basin. Also, a transformation $\prod_{\text{spatial}}$ is defined to extract objects the areas of objects such as DEMs.

`access` $W = \{x$ `in` *DEM* $\mid \prod_{\text{spatial}}(x)$ *intersect* $\prod_{\text{spatial}}(Manaus)\}$

Further assume that *union* defined on *DEM* produces a single DEM from a set of DEMs. We can produce the DEM for the area Manaus simply with:

*Manaus-dem* `= apply` *DEM.union* `to` $W$ □

# 7 Implementation and Applications

We are currently working with a group of EOS earth science investigators to jointly implement and apply a high-performance version of an MDBS that incorporates the data model and high-level languages described above. This system is intended to support large-scale earth science investigations by complementing the EOS information system (EOS-DIS) [6] in terms of support for high-level modeling and data management. As noted above, MDBS is intended to facilitate the iterative development of models in data-intensive and computationally intensive applications. The project represents a major extension of a 3-year, interdisciplinary NSF-supported research project on scientific databases, which has resulted in the design of the data model and languages [22, 21].

MDBS is being developed with the specific goal of solving several major EOS-scale problems in earth science, including the routing of water and sediment down the entire Amazon basin, the integration of GCM's and regional hydrological models, and the modeling of snowmelt and the routing of runoff from regional snowpacks. It is also designed to be generally applicable to a large class of scientific modeling problems that involve both data-intensive and computationally intensive aspects. Based upon our data model, we are constructing a large lattice of *representational domains* in order to organize the data, algorithms and computational modeling activities of the applications scientists. A simple prototype of the data model is currently being built. It implements the data model and MDBL based on the POSTGRES database management system [17, 23]. The database contains the datasets from applications menetioned above and is distributed over a network. A virtual file system (not described here) is employed to provide network transparency.

Based on this model and the set of R-domains, the system will support a visual interface that implements, in part, the MDBL languages. The system will provide easy access to distributed databases of datasets and algorithms, and will permit the efficient reuse of code for building new algorithms. Support will also be provided for *lineage* and *error tracking*, for the concepts of *model* and *project*, and for "model debugging". In order for systems such as MDBS to be acceptable to scientists it is critical that they possess a high degree of performance. Such performance will stem in part from parallelization and also from the provision of high-level modeling tools that are based on the data model and permit scientists to construct creative solutions to apparently intensive computational problems.

# 8 Conclusions

We have designed an advanced and comprehensive data model and a high-level database and modeling language (MDBL). The data model is based on a simple characterization of scientific activity that captures the essence of scientific modeling and database activities in terms of four fundamental sets of activities. These activities are defined in relation to large lattices of *representational domains* (R-domains) that are employed to organize scientific knowledge, data and algorithms. The data model separates the specification and representation of R-domains of elements, and allows multiple representations that are explicitly controlled by the user. Datasets and transformations are organized in terms of an extensible lattice of abstract and concrete R-domains. In particular, the data model allows the easy definition and manipulation of large numbers of domains of spatio-temporal elements. MDBL is a high-level, largely declarative language that is also based on our characterization of science and provides easy expression of the four classes of scientific activities in relation to R-domains. MDBL permits the easy integration of modules that support scientific modeling and database activities. The data model and the languages are illustrated by examples from EOS earth-science applications.

# References


[1] S. Abiteboul and C. Beeri. On the power of languages for the manipulation of complex objects. Technical Report No.846, INRIA, May 1988.

[2] S. Abiteboul and P. C. Kanellakis. Object identity as a query language primitive. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 159–173, 1989.

[3] J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, November 1983.

[4] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In *Proc. First Int's Conf. on Deductive and Object-Oriented Databases*, pages 40–57, 1989.

[5] A. K. Chandra and D. Harel. Computable queries for relational data bases. *Journal of Computer and System Sciences*, 21(2):156–178, 1980.

[6] J. Dozier. Looking ahead to EOS: The earth observing system. *Computer in Physics*, May/June 1990.

[7] EOS: A Mission to Planet Earth, NASA, Washington, D.C., 1990.

[8] R. Hull. Relative information capacity of simple relational schemata. *SIAM Journal on Computing*, 15(3):856–886, 1986.

[9] R. Hull and R. King. Semantic data modeling: Survey, applications, and research issues. *ACM Computing Surveys*, 19(3):201–260, September 1987.

[10] R. Hull and J. Su. Untyped sets, invention, and computable queries. In *Proc. ACM Symp. on Principles of Database Systems*, pages 347–359, 1989.

[11] Bulletin of the Technical Committee on *Data Engineering*, 16(1), March 1993. (Special Issue on Scientific Databases).

[12] C. Lecluse and P. Richard. Manipulation of structured values in object-oriented databases. In *Proc. 2nd Int. Workshop on Database Programming languages*, pages 113–121, 1989.

[13] C. Lecluse and P. Richard. Modeling complex structures in object-oriented databases. In *Proc. ACM Symp. on Principles of Database Systems*, pages 360–368, 1989.

[14] C. Lecluse, P. Richard, and F. Velez. $O_2$: An object-oriented formal data model. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 424–433, Chicago, June 1988.

[15] R. H. Güting. Geo-Relational Algebra: A Model And Query Language For Geometric Database Systems. In *Conference on Extending Database Technology (EDBT '88)*, 1988.

[16] R. H. Güting. Modeling non-standard database systems by many-sorted algebras. Technical Report 225, Fachbereich Infromatik, Universität Dortmund, 1988.

[17] L. A. Rowe and M. R. Stonebraker. The POSTGRES data model. In *Proc. Int. Conf. on Very Large Data Bases*, pages 83–96, 1987.

[18] M. Scholl and A. Voisard. Thematic map modeling. In *Design and Implementation of Large Spatial Databases (SSD'89)*, pages 167–192, 1989. Lecture Notes in Computer Science No. 409, Springer-Verlag.

[19] Spatial Data Transfer Survey, U.S. Geological Survey, 1992.

[20] T. R. Smith, J. Su, D. Agrawal, and A. El Abbadi. Database and modeling systems for the earth sciences. In IEEE [11]. (Special Issue on Scientific Databases).

[21] T. R. Smith, J. Su, D. Agrawal, and A. El Abbadi. MDBS: A modeling and database system to support research in the earth sciences. Technical report, Dept. of Computer Science, UCSB, 1993.

[22] T. R. Smith, J. Su, T. Dunne, A. El Abbadi, and D. Agrawal. Requirements for modeling and database systems supporting investigations of large scale flows in the amazon watershed, 1993. (In preparation).

[23] M. Stonebraker, L. A. Rowe, and M. Hirohama. The implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):125–142, March 1990.

[24] J. D. Ullman. *Principles of Database Systems (2nd edition)*. Computer Science Press, Potomac, Maryland, 1982.

[25] M. F. Worboys. A generic model for planar geographical objects. *International Journal of Geographical Information Systems*, 6(5):353–372, September/October 1992.

# APPENDIX I: Definitions of Spatio-temporal Domains

We now define the remainder of the spatio-temporal domains that were listed in Section 4.

The next domain that we define, *SimpleArcs* may be viewed as being constructed by the application of a value constraint to the elements of the domain *Arcs*.

**abstract domain:** *SimpleArcs*
**inherits:** *Arcs*
**constraints:** $\forall x \in \textit{SimpleArcs}, \textit{selfintersect}(x) = false$

The *inherits* statement indicates that *SimpleArcs* inherit both the transformations and the concrete representations of *Arcs*.

**abstract domain:** *Loops*
**inherits:** *Arcs*
**constraints:** $\forall x \in \textit{Loops}, \textit{extremes}(x) = \emptyset$

**abstract domain:** *LoopSets*
**concrete domain:** *LoopSets*$\star$
**structure:** $\{\textit{Loops}\}$

**abstract domain:** *Strands*
**inherits:** *Arcs*
**constraints:** $\forall x \in \text{Strands}, x \notin \textit{Loops}$

**abstract domain:** *SimpleLoops*
**inherits:** *SimpleArcs*
*Loops*
**operations:** *inside* : *Points* $\times$ *SimpleLoops* $\rightarrow$ *Booleans*
*inside* : *SimpleLoops* $\times$ *SimpleLoops* $\rightarrow$ *Booleans*

The transformation *inside* returns *true* if the point is inside a simple loop, or one simple loop is entirely inside the other.

**abstract domain:** *SimpleStrands*
**inherits:** *SimpleArcs*
*Strands*

**abstract domain:** *SimpleAreas*
**concrete domains:** *SimpleAreas$\star$LoopSets*
**operations:** *boundary* : *SimpleAreas* $\rightarrow$ *LoopSets*
*centroid* : *SimpleAreas* $\rightarrow$ *Points*
*area* : *SimpleAreas* $\rightarrow$ *Reals*
*perimeter* : *SimpleAreas* $\rightarrow$ *Reals*
*adjacent* : *SimpleAreas* $\times$ *SimpleAreas* $\rightarrow$ *Booleans*
*union* : *SimpleAreas* $\times$ *SimpleAreas* $\rightarrow$ *SimpleAreas*
*intersection* : *SimpleArea* $\times$ *SimpleArea* $\rightarrow$ *SimpleArea*
*components* : *SimpleArea* $\rightarrow$ *Integer*

The operation *components* returns the number of connected components.

**concrete domain:** *SimpleAreas$\star$LoopSets (SALS)*
**implements:** *SimpleArea*
**structure:** $\{$*SimpleLoop*$\}$
**constraints:** $\forall x \in iboxSALS \; . \; (l_1, l_2 \in x$ and $l_1 \neq l_2) \Rightarrow$
$($intersection$(l_1, l_2) = \emptyset$
and inside$(l_1, l_2) =$ ‘false’
and inside$(l_2, l_1) =$ ‘false’$)$
(Ensures simple loops do not overlap or nest inside each other).
**operations:** *boundary, centroid, area, perimeter adjacent,*
*union, intersection, components.*

**abstract domain:** *SimpleRegions*
**inherits:** *SimpleAreas*
**constraints:** $\forall x \in$ *SimpleAreas*$,$ *components*$(x) = 1$

We have only considered the simplest kinds of areal objects, namely those without holes. More complex domains may be modeled in a similar way to the exemplars above.

# APPENDIX II: An Example of Application

In the application of Amazon drainage basin that we are working with, one major program is the routing of surface water through Amazon channel network. In this section, we focus on an example which is a simplified version of the routing surface water problem. We shall first describe the computation in the terminologies of domains, elements, and transformations in our data model. We then present a sequence of operations in English that are needed for solving the routing problem. This sequence of operations also represents a "typical" sequence of steps that a scientist would like to perform on databases. Finally, we provide the operations in MDBL to illustrate that such operations could be expressed using MDBL in a simple manner.

The example is simplified in terms of both the number (and sizes) of the datasets involved and the computation. Briefly speaking, the sequence of the computation involved in the example is to first choose the DEMs of interest (intersecting with the area Manaus) and to combine them into one single DEM. Then, rainfall data inside the relevant area for certain specific time period are retrieved and interpolated over the DEM. We use *find-storms* to extract all storms from the rainfall data for each hour. Finally, the DEM will be used to generate a "equal travel distance map", which contains the information about the water traveling time from each point in the DEM to the *mouth* of the river (the lowest point in the area). The hydrograph of the discharge of water at the mouth as a function of time instants is then computed. Figure 1 shows the datasets and the operations involved. A sequence of operations in English is shown in Figure 2.

Suppose we already have domains *DEM* and *RainFall* defined and `Manaus` is a variable holding a polygon object which represent the Manaus area. The operations can simply be expressed as the following MDBL commands.

1. ```
   access X = { D in DEM
   where spatial-projection(D) intersect spatial-projection(Manaus)}
   ```

   The function *spatial-projection* projects objects to their spatial components and *intersect* is a spatial predicate.

2. ```
   Manaus-dem = apply DEM.union to X
   where spatial-projection(Manaus-dem) = spatial-projection(Manaus)
   ```

   *DEM.union* is a transformation defined on the domain *DEM*, which combines many DEMs into one.

3. ```
   apply 3dplot to Manaus-dem
   ```

   The operation *3dplot* displays the DEM as a three dimensional mesh. Such display functions are provided from other tools (such as MATLAB) and "registered" in an MDBS as transformations. The abstract/concrete domains in our model makes such integration of tools relatively easy.

4. ```
   create concrete domain Slope
   where abstract domain = Slope
         structure = {[Point, [1..4]]}
         transformation = dem2slope: DEM -> Slope is <fun-exp>
   ```

   The function expression `<func-exp>` defines the body of the transformation.

5. ```
   create element Manaus-slope in Slope
   where value = apply dem2slope to Manaus-dem
   ```

6. ```
   create concrete domain Binary
   where abstract domain = ETDM
   ```

```
structure = raster
transformation = dem2etdm:DEM -> ETDM is dem2etdm
transformation = compute-discharge:
                    (ETDM, {Storm})-> Hydrograph is discharge
```

Here the concrete domain is represented in binary format as “black boxes”. The transformations are thus provided as executable programs.

7. `create element Manaus-etdm in ETDM`
   `where value = apply dem2etdm to Manaus-dem`

8. `create element Y in rain_tpr where value = <query-expression>`

   The query expression is shown in Example 6.2.

9. `create element Z in RainFall.RF-PxR`
   `where value = apply rain-interpolate to Y`

10. 
```
create concrete domain S-PxR
where abstract domain = Storm
      super-domain    = RainFall.RF-PxR
      transformation  = create-storms: RainFall -> {Storm}
                         is <fun-exp>
```

   Since the type for the new domain is the same as its super-domain, the type specification can be omitted. The function expression can be easily written; a simpler version is shown in Example 5.8.

11. `create elements in Storm where values = apply create-storms to Z`

12. `Manaus-discharge = apply compute-discharge to (X, W)`
    `where W in Storm`

13. `apply 2dplot to Manaus-discharge`