# VCME: A Visual Interactive Environment for Computational Modeling Systems*

Yongmao Chen† Terence R. Smith Yuan-Fang Wang Amitabh Saran‡

Department of Computer Science
University of California
Santa Barbara, CA 93106, USA
e-mail: {yongmao, smithtr, yfwang, saran}@cs.ucsb.edu

* Work supported in part by NASA grant NAGW-3888.
† Work supported in part by NASA grant NGT-30159.
‡ Currently working at Hewlett Packard-ISO, India. Email : saran@india.hp.com

**Abstract**

In previous work, we have described the concept of a *computational modeling system* (CMS) that is based on a comprehensive conceptualization of scientific modeling activities. A CMS involves an integrated computational modeling environment (CME), that embodies this conceptualization and in which a scientist may carry out, at an appropriate level of abstraction, any operation involved in the process of symbolic model construction. The key contribution of the current paper is the translation of the CME into graphic terms and, in particular, the design and implementation of a visual language for supporting the top-down construction of scientific models. We describe the design and implementation of a *visual* CME (VCME) for our CMS. The VCME supports the construction and use of representations of any modeling concept that a scientist may require in the iterative process of building symbolic models of phenomena. In particular, the VCME supports a new computational approach to the top-down construction of computational models. The *flowchart editor*, which is the main component of the user interface in supporting such constructions, also supports the bottom-up process of model construction. A unique feature in our system is its ability to provide generic support for resources under a standard, coherent framework that makes easy integration of arbitrary sets of data and tools, *distributed over the network*, and provide support for their interoperability at the CME level. Our design and implementation integrates multiple user interface modes, code generators, visual language, data manipulations, computational model processing, and communication with other tools. This provides the user with a powerful software development environment that facilitates the complex procedures in the construction, testing, and application of scientific models.

# 1 Introduction

A major challenge for the computational and information sciences is to provide integrated support, based on digital technology, for the modeling activities of scientists involved in numerically-intensive and data-intensive applications. In response to this challenge we have developed the notion of a *computational modeling system* (CMS) [1].

## 1.1 Computational Modeling System (CMS)

A CMS involves two major components. The first component is an integrated *computational modeling environment* (CME) in which a scientist may represent any symbolic modeling activity. In particular, a CME is intended to provide an environment in which a scientist may engage in the iterative process by which symbolic models of phenomena are constructed, evaluated, and applied. The second component involves appropriate computational support for the modeling activities represented in the CME. Apart from supporting the construction and manipulation of scientific modeling concepts at appropriate levels of abstraction, such support should permit the easy integration of arbitrary sets of data and tools, distributed over the network. We term this component the *distributed resource access system* (DRAS).

There are three general requirements for a CME. First, it should be based on a general and relatively complete model of the process of constructing, testing, and applying symbolic models of phenomena. Second, the modeling operations carried out in the CME should be provided with efficient computational support from an extensible set of computational modules in the CMS. Third, the CME should be represented to the scientific modeler in terms that are simple and expressive and that conceal irrelevant computational details.

The focus of the current paper is on the graphical representation of the CME and, in particular, on a visual language that supports the construction of a relatively simple conceptualization of *models*. Graphical support in a CME is an important issue since the sets of concepts, the representations of these concepts, and the manipulation of the representations that are employed in scientific modeling enterprises are typically large in number and frequently complex. Furthermore, models of phenomena that are constructed and manipulated on the basis of these concepts are also typically large and complex. Graphical representations of the constructions and manipulations in a CME offer the possibility of minimizing much of this complexity from the viewpoint of a user and provide a framework to access network-wide resources.

We term the visual representation of a CME the *visual computational modeling environment* (VCME) of the CMS. A key issue in the design of an appropriate VCME is to provide scientists with a visual representation of the modeling process, expressed at the conceptual level of the modeling application.

There are two main contributions of the paper. The first is the design and implementation of a VCME that rep-

resents a visual translation of a CME based on a general and relatively complete model of the scientific modeling process. The second contribution is a visual language that is intended to facilitate the top-down construction of models of scientific phenomena. The class of models that are supported by this visual language is relatively broad and is based on representations of an important class of concepts that are representable in a CME.

## 1.2 Related Work

It is useful to review briefly the nature of support for scientific modeling activities provided by current graphics and visual computing technology. Such support may be classified into three major categories. The first category involves *visual languages* that may be employed for representing programs, data, and the structure and behavior of complex systems. The second category involves specific *graphical interface tools* that have been developed to support various programming environments and tailored to support scientific modeling activities. Finally, the third category pertains to mechanisms and tools that provide access to *distributed resources*.

The class of visual languages [3, 4] includes languages for handling visual information, such as GRAIN [5]; for supporting visual interaction, such as PSQL [6] and ICDL [7] and HI-VISUAL [8]; for programming with visual expressions, including data flow diagram languages that provide views of a computation in terms of the flow of data from one filter to another, such as LabVIEW [9] for science and PROGRAPH [10] for general-purpose; and for dealing with visual objects that are themselves visually represented, which are often termed "iconic visual information processing languages" [4]. However, these visual languages are essentially programming languages that may be employed in writing programs that support specific applications, and there is no database support.

There are also visualization tools that have been developed for the graphical representation of specific computing environments. Examples of such systems include the graphical user interface to Khoros, which is a software environment for supporting research in image processing [11, 12, 13, 14]; scientific programming systems like AVS [15] and IBM Data Explorer (IBM DX) [16, 17, 18], which are similar to the Khoros system; and Tioga [19], which is a visual programming system for scientific DBMS management applications.

With respect to providing a graphically-represented environment for the support of scientific modeling, there are several problems associated with the languages and systems listed above. Most importantly, neither the languages nor the systems are based on general scientific modeling principles. At best, some of the systems, such as Tioga, IBM DX, are based on partial models of specific modeling activities.

Other problems include a typical lack of support for high-level data and computation abstraction, and particularly for abstractions relating to the concept of a "model". In existing systems, the user must typically construct application models using file-based data and functions. In IBM DX, for example, the data model is based on low-level file, such

as image data, or map data. Hence the concept of a model is hidden in the low-level data structures and files. But data in VCME is supported through a hierachical object-oriented model represented by abstract and concrete R-structures and their instances.

For model construction, Khoros and IBM DX allows users to group modules to form higher-level modules, referred to as bottom-up construction. VCME, however, supports two ways of doing this, bottom-up and top-down construction. The top-down is a more natural way to design a computational model for scientists and engineers.

Most current systems also lack the full range of computational support necessary for scientific modeling activities. The languages, for example, provide inadequate database support and do not provide support for distributed model development. IBM DX provides support for interactive visualization and analysis of both observed and simulated data. Our system not only supports data interaction, but also provides a interactive model for executing modules at local and remote locations. Most systems [21, 22, 23, 24, 25] are either based on evolutionary extensions to existing database technology, or require a considerable amount of custom coding tailored to the application [35, 34].

A promising and general approach to integrating distributed data is offered by the *Common Object Request Broker Architecture* (CORBA) [37] of the Object Management Group (OMG) [36]. The use of *Object Request Broker* (ORB) to bind components together offers substantial savings in design, implementation, and maintenance efforts. The functionality offered by CORBA, however, is not yet complete. Current efforts are focussed on value-added services to augment the basic ORB. Web-based tools are increasingly becoming common. However, most ([29, 28, 31]) provide access to remote data but none for external tools and legacy systems. Then there are systems that deal exclusively with the integration of tools, without a conceptual basis for organizing application data. PCTE[33], for instance, provides a set of C and Ada specifications that can be used by tools for process control, distribution and accounting. In QUEST[32], the system has been made to support a defined set of tools that assist in analyzing geophysical datasets. Finally some complete systems like SHORE[38], which provides a merger of object-oriented and file system technologies, and Harvest[27] which also provides object-oriented extensions to define type hierarchies and complex data access methods, exist but they lack a unified computational environment that supports the iterative process by which symbolic representations of application phenomena are constructed, evaluated and applied, features that the CME and DRAS are well capable of supporting.

The paper is structured as follows. In order to make the paper relatively self-contained, we first provide a concrete example of the scientific modeling process in Section 2. This example is employed in the remainder of the paper. Although we have described elsewhere [1] a simple, general, and relatively complete basis for the modeling operations in a CME, we briefly summarize this basis for the sake of completeness. Section 3 then describes the functionality, architecture, and application of a VCME that embodies this basis. Finally, we describe the design and implementation

of the visual language for the top-down construction of scientific models in Section 4, Distributed Resource Access System (DRAS) of VCME in Section 5, Code Generation specification in Section 6, and conclude with details of a specific implementation of a VCME in the following section.

# 2 Support for Modeling in a Computational Environment

We may characterize the process of constructing and investigating *symbolic models* of phenomena as one in which concepts that characterize both the phenomena under study and the process of modeling itself are represented and manipulated in a symbolic manner. The CME of a CMS is an environment in which both concepts can be achieved.

A reasonable approach to designing and implementing a CME is to:

1. characterize the nature of the symbolic modeling process in terms of a conceptual basis that is reasonably "complete" with respect to a broad range of modeling activities;
2. design a CME that incorporates this conceptual basis;
3. provide efficient support based on digital technology for such environments;
4. design and implement graphical representations of a CME that facilitates its use in complex modeling applications.

Before discussing graphical representations of a CME, we provide a brief overview of a conceptual basis for a CME. Further details of this treatment of a CME may be found in [1].

We motivate this overview by first describing a specific modeling application drawn from the hydrological sciences. This is an instructive example, since we believe that a CME and its visual representation should be capable of representing the full range of modeling activities embodied in the example.

## 2.1 An Example of Scientific Modeling

In Figure 1, we represent a simplified view of some of the entities and procedures employed in constructing a model of a hydrological phenomenon. The goal of the modeling procedure is to predict the flow of water at various locations in a river basin. For simplicity, we have indicated only one of the many feedback loops involved in the modeling process. In Figure 2, we provide brief descriptions of the modeling operations that are implicit in Figure 1. Important classes of modeling activities include:

1. the extraction of relevant information from "datasets";

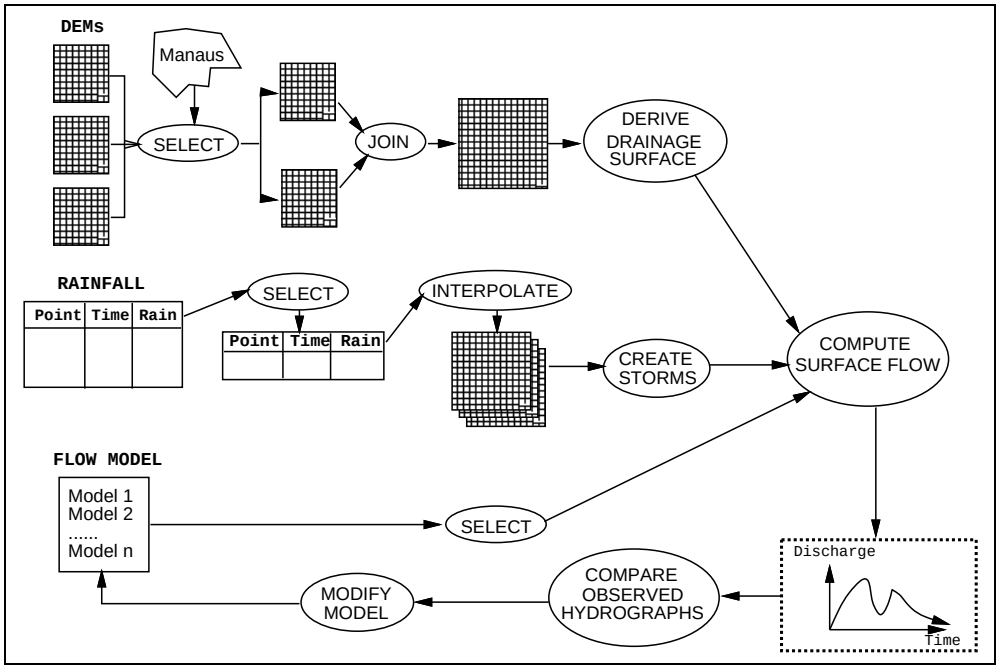


Figure 1: A Graphical Representation of a Simple Modeling Process

2. the construction and application of "models" of the flow of water;

3. the evaluation of the modeled flows;

4. the iterative improvement of the components of the modeling process;

5. the communication of the results of the modeling process.

It is reasonable to interpret these modeling activities in terms of procedures by which investigators represent and manipulate a large body of modeling concepts; choose from among alternative concepts and representations; and seek new concepts and representations. We briefly illustrate this interpretation in terms of the hydrological modeling example of Figure 2.

In relation to the example, scientific issues concerning the choice or discovery of concepts and representations for the concepts arise in steps 3, 7, 9, 10, and 14. For example, a question of some interest for a scientific investigator concerns the manner in which the surface represented in the display of step 2 determines the magnitude and direction of surface runoff. To answer this question, an investigator may employ concepts that characterize the nature of the surface and the interactions between the surface and the flow of water. Examples of such concepts include *surface slope*, *channel segment*, *channel segment contributing area*, and *drainage basin*. If these concepts are to have relevance

1. Get the digital elevation models (DEMs) that intersect the Manaus area.
2. Join this set of DEMs into single DEM and display it.
3. Create a general class of representations for river channel segments and contributing slopes of the drainage system implicit in a DEM.
4. Create and store an instance of this class of representations from the DEM constructed for the Manaus area and store
5. Find time-slices of rainfall from hour 1 to hour 12 on January 21 1989 for rain-fall records within the Manaus area.
6. Create a rainfall raster for each of the previous datasets using a rainfall interpolation routine.
7. Create a general class of representations for rainstorm events.
8. Find and store all rainstorm events that can be found in the rainfall raster just created.
9. Create a general class of representations for the flow vectors of water on the drainage surface.
10. Construct a model for predicting the surface flow vectors of water over a drainage surface in response to some rainstorm event.
11. Apply the flow prediction model to the specific drainage surface and rainfall event just constructed.
12. Using the map of predicted surface flow vectors, compute the hydrograph that would occur at the downstream end of the highest order channel-segment on the drainage surface.
13. Display a plot of this hydrograph and compare statistically the predicted and observed hydrographs.
14. Create a class of modeling-schemas that encapsulate the sequence of operations in steps 1-13 above.
15. Run this schema iteratively using variants of the flow model from step 10 until the predicted and observed hydrographs are in reasonable agreement.

Figure 2: A Hydrological Modeling Process Represented in Natural Language

in a modeling process in which observations on landsurfaces are represented by *digital elevation models* (DEMs), it is necessary that they possess concrete representations that can be extracted from DEMs.

There are, however, many modeling situations for which neither a "standard" set of concepts nor a "standard" set of representations for concepts has been defined. An important activity in such cases may be the discovery of an appropriate set of concepts and representations. Hence in steps 3-4 of our example, the investigator may (1) create new classes of representations for concepts relating to landsurface characterization; (2) extract instances of such representations by transforming DEMs; and (3) evaluate the representations using visual transformations. A transformation is a mapping from one set of representation domains (R-domain) to other set of R-domain. Similar remarks apply with respect to observations on rainfall events and flow events (steps 7-10). In relation to step 10, for example, an investigator may choose to characterize surface water flows in terms of *surface flow vectors* involving representations of *time*, *location*, and the *magnitude* and *direction* of flows. For the purpose of predicting flows over landsurfaces represented in terms of *channel segments* and *channel segment contributing area*, it may be appropriate to construct classes of flow representations in which *surface flow vectors* are decomposed into *channel segment flow vectors* and *overland flow vectors.*

For reasons of convenience, the investigator may wish to employ concepts and representations of concepts that describe the *process of modeling itself.* Such representations could be manipulated during the process of model evaluation and iterative improvement. In particular, once a representation for the entire process has been constructed, it may then be modified, versioned, transformed, and executed as a unit, as in steps 14-15 of Figure 2.

A key element in many modeling situations involves the application of *transformations* to concept representations. In terms of our example, the use of transformations arises in steps 2, 3, 4, 5, 6, 11, 12, 13, and 15. In steps 2 and 4, for example, we note that the various procedures on the DEMs may be interpreted as transformations in which one set of representations of landsurfaces is mapped into other sets. The *display* command in step 2, in particular, may be viewed as a transformation that maps the representation of a DEM into a representation that is visually meaningful. In step 11, it is necessary to select and apply *solution procedures* to the representation of the flow generating process. The effect of applying such procedures may be interpreted in terms of transformations that map representations of land surfaces, rainstorm events, initial flows over the surfaces, and the flow-generating models into representations of flows over surfaces at various times.

Information may be extracted from such flow representations by the application of further transformations. It may, for example, be desirable to compute representations of *hydrographs* at various locations on *channel segments* (step 12). The visualizations and comparisons of the observed and computed *hydrographs* that occur in the model evaluation step (13) may be interpreted as transformations. The comparison of observed and computed hydrographs may, for example, be viewed as a transformation from pairs of representations of hydrographs into representations of statistical measures.

## 2.2 A Conceptual Basis for the Symbolic Modeling Process

We believe that it is critical to base computational support for scientific applications, such as the example illustrated in section 2.1, on the basis of a sound model of the scientific modeling process. We now present a brief description of such a model.

A key aspect of our model of scientific activity is its focus on the nature, representation, and use of *concepts* as the central components of symbolic modeling activities. In particular, this approach leads directly to the design and construction of a CME in which it is possible to construct relatively high-level expressions for any symbolic modeling activity. The approach places no constraints on the sets of concepts that may be defined nor on how they may be represented. Furthermore, it does not constrain the order in which scientific modeling activities are carried out.

Scientific modeling is a complex enterprise comprising a large range of activities. Apart from observation and measurement-related activities, *the* core activity of modeling activities is the construction and manipulation of *symbolic representations of phenomena*. As we argue in [1] it is natural to view the process by which symbolic models of phenomena are constructed as a self-reflective process in which an enormous space of concepts, representations of concepts, and transformations between the representations are explored. More specifically, the process may be viewed as one involving the

1. the choice or discovery of sets of concepts;

2. the choice of representations for the concepts, the definition of associated transformations, and the creation of representations of concept instances;

3. the application of transformations to representations of concept instances;

4. the construction of interpretative mappings from symbolic representations into "phenomena".

The viewpoint also captures the fact that scientists typically employ concepts relating to the *process of modeling itself*, as well as concepts relating to the phenomena under investigation. A good example of such a concept is that of a *model*. Although these concepts play an important role in such activities as the evaluation of models and exchanges of information concerning models, they are poorly supported in current computational environments. Not only does a CMS permit support for the concept of a model, but it also permits the construction of many classes of models from previously defined concepts.

### 2.2.1 Representational Structures and Scientific Modeling

It is possible to construct a relatively simple conceptual basis of fundamental activities that support scientific modeling [1]. This conceptual basis involves the idea of *representational structures* (R-structures). R-structures provide a language for constructing and manipulating *representations of concepts* and are the foundation on which the CME of a CMS is constructed.

We define a *representational structure* (or *R-structures*) for a concept to be a triple $[\mathcal{D}, \mathcal{T}, \mathcal{I}]$ in which (1) $\mathcal{D}$ is the *representational domain* (*R-domain*) of the R-structure; (2) $\mathcal{T}$ is a set of *transformations* that may be applied to $\mathcal{D}$; and (3) $\mathcal{I}$ is a finite subset of instances of $\mathcal{D}$. The R-domain $\mathcal{D}$ of an R-structure contains a set of *representations* for all instances of the concept. Since an R-domain $\mathcal{D}$ will typically contain a large or even infinite number of such representations, an element of $D$ will typically be specified in terms of a schema that defines the set of representations in some implicit manner. $\mathcal{T}$, or the *transformations* of the R-structure, is a set of representations of transformations that may be applied to the representations in the R-domain $\mathcal{D}$. Any transformation $t$ in $\mathcal{T}$ is the cartesian product of $D_1$,...,$D_n$ to cartesian product of $D'_1$,...,$D'_m$. Here, $D_i$ ($i = 1, n$) and $D'_j$ ($j = 1, m$) are R-domain. $\mathcal{I}$ is a finite set of representations from the R-domain that are given in *explicit form* and that have particular significance in the modeling enterprise. In general, R-structures are given a name so that they may be referred to as entities.

We illustrate these ideas with an R-structure for representing the concept of *Polygons*. A representation for *Polygons* may be constructed in terms of an R-structure named **Polygons::Points**.

Figure 3 illustrates the R-structure of **Polygons::Points** that contains three parts, $[\mathcal{D}, \mathcal{T}, \mathcal{I}]$.

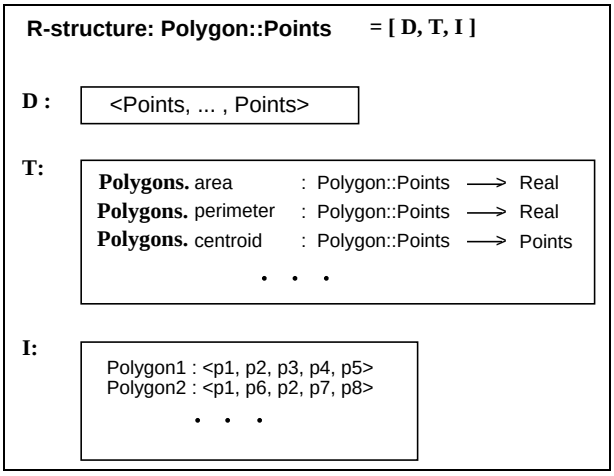


Figure 3: An example of concrete R-structure of **Polygons::Points**

The R-domain of **Polygons::Points** contains *concrete* representations of polygons as sequences of points $\langle p_1, ..., p_n \rangle$ that may be interpreted as the vertices of the polygon. The points $p_i$ $(i = 1, n)$ may be viewed as representations from the R-domain of an R-structure **Points**. The notation **Polygons::Points** indicates that the concrete representation of the concept of *Polygons* is in terms of concrete representations of the concept of *Points*. There may be multiple concrete representations of concepts. For example, *Polygons* may also be represented in terms of *Linesegments* or *Half-planes*.

In order to make this representation of the concept *Polygons* accord with our intuition concerning polygons, it is necessary to associate with the R-domain of **Polygons::Points** expressions that represent constraints, such as "*no two edges of the boundary intersect, except at their end points*". The set of transformations $\mathcal{T}$ on the representations of polygons might include, for example, **Polygons.***area*, which maps a representation of a polygon into a representation of its *area*, and other transformations such as **Polygons.***perimeter* and **Polygons.***centroid*. Finally, we might wish to construct a specific set $\mathcal{I}$ of polygons that are frequently encountered in some modeling activity.

As noted in [1], it is of value to differentiate between *concrete R-structures* and *abstract R-structures*. The former correspond to the concrete, manipulable representations of concepts. Since for many concepts there are multiple, equivalent concrete representations, it proves useful to define an *abstract* R-structure over the set of all equivalent concrete R-structures. Such abstractions facilitate the construction of new concrete R-structures and may be used to support the notion of inheritance.

If one views R-structures as representations of phenomenological or methodological concepts, then the process of constructing symbolic models of phenomena may be viewed as one in which a particular set of R-structures is inductively constructed and explored. R-structures are "evaluated" in terms of interpretations that map R-domain elements and the associated transformations into either the phenomena of interest or the process of modeling. Hence we may characterize the process by which scientific representations of phenomena are constructed and evaluated *in*

*terms of R-structures* as a process in which scientists (1) construct, evaluate and apply collections of R-structures for modeling both the phenomena in specific domains of application and the phenomena of the modeling process itself; and (2) construct *specific instances* of R-domain elements and apply sequences of specific transformations to sets of instances of R-domain elements. "Useful" R-structures are retained and their value communicated to other scientists while others are abandoned.

R-structures may be constructed that permit scientists to represent, as first class entities, any reasonable concept that is useful in the modeling process. We may, for example, construct specific R-structures whose R-domains contain representations of primitive "abstract" entities, such as integers, real numbers, boolean values, and character strings; abstract entities, such as simple and complex geometrical figures; abstract mathematical models, such as partial differential equations; "empirically derived" entities, such as datasets obtained by various observational means; transformations (e.g. procedures, algorithms, and statistical operations); and scientific *modeling schemas* that are defined in terms of the representations and transformations of other R-structures and scientific *projects*.

### 2.2.2 Supporting the Concept of *models* and *Projects*

Two important, high-level concepts that arise in scientific modeling applications and that may be represented in terms of R-structures are *models* and *projects*. We briefly indicate how such representations may be constructed. We first note, however, that our definition of the concept of a model is not intended to be complete in any sense, but is aimed at capturing the essence of a significant class of scientific models. A great advantage of the approach supported by R-structures is that the definition of concrete representations of some concept in no way proscribes the use of alternative representations.

Out notion of the concept of a model is intended to capture the idea of sequences of modeling operations that transform sets of input representations into sets of output representations. Such sequences are illustrated in Figures 1 and 2. Furthermore, we generalize this notion of a model to one of *modeling schemas* in which one may represent any symbolic modeling operation. Hence a modeling schema may include not only the operation of applying transformations to concept representations, but also modeling operations such as the creation of R-structures or the accessing of R-structure instances. Figure 4 illustrates the model as a R-structure. The R-domain of the model is directed graph which represents the computation process. The set $\mathcal{T}$ of transformations associated with some R-structure **Modeling-Schemas::Graphs** includes operations that permit one to *construct*, *run*, *test*, and *modify* instances of modeling schemas in an interactive manner. In Figure 4 the trasformation **Model.***run* maps graph with some instances od R-domains to other instances of domains. **Model.***run* does not change the graph, but only derives the results from the computation graph (a flowchart) with the input instances (called instatiation). Examples of transformations related to the modifi-

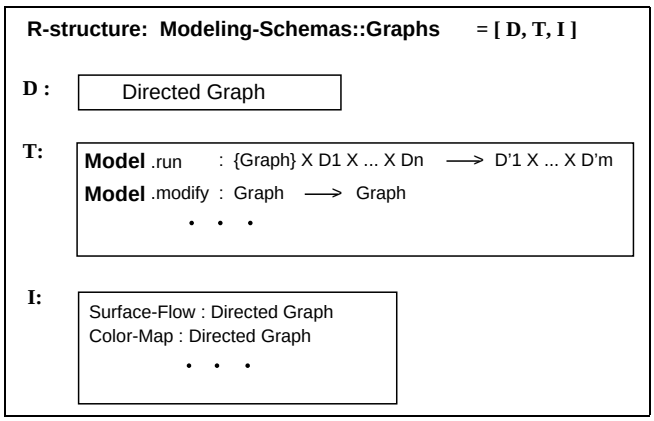


Figure 4: Model is an R-structure

cation of modeling schemas include transformations that permit one to concatenate two modeling schemas or to set break points for testing purposes. Examples of transformations related to the running of modeling schemas include *run*. The set $\mathcal{I}$ of the R-structure contains explicit representations of specific instances of the modeling schema.

Figure 5 is an instance of the R-structure **Modeling-Schemas::Graphs**. In Figure 5 we illustrate such a modeling schema for the hydrological modeling example. This modeling schema is intended to characterize a simplified version of the modeling schema of Figure 2 in which the only modeling operations are the *application* of transformations and the *accessing* of R-domain elements. In particular, this modeling schema does not involve the *create* operation, and the graph of the modeling schema has no cycles.

It is natural to represent modeling schemas as graphs in which the nodes may represent instances of R-domain elements or transformations and the edges represent data flows between the nodes. An instance of a modeling schema may be viewed as a representation of a specific modeling process. Such representations permit one to represent a specific modeling process or model as a unit, and to facilitate the iterative creation, testing, modification, and applications of models.

The concept of a *project* is currently the highest level concept that we represent in terms of an R-structure. It permits one to define the idea of subprojects and the interrelationships between related projects in terms of the sharing of various R-structures and their component elements. A *project* may be represented in terms of a subset of the available R-structures that is sufficient for the iterative modeling and database activities in some domain of scientific application.

The R-structure, **Projects**, incorporates an R-domain $\mathcal{D}$ whose elements are tuples $[e_1,\ e_2, ...]$ in which $e_1$ is a any admissible subset of R-structures (except *projects*) and $e_2$ is a subset of instances of modeling schemas. Other elements

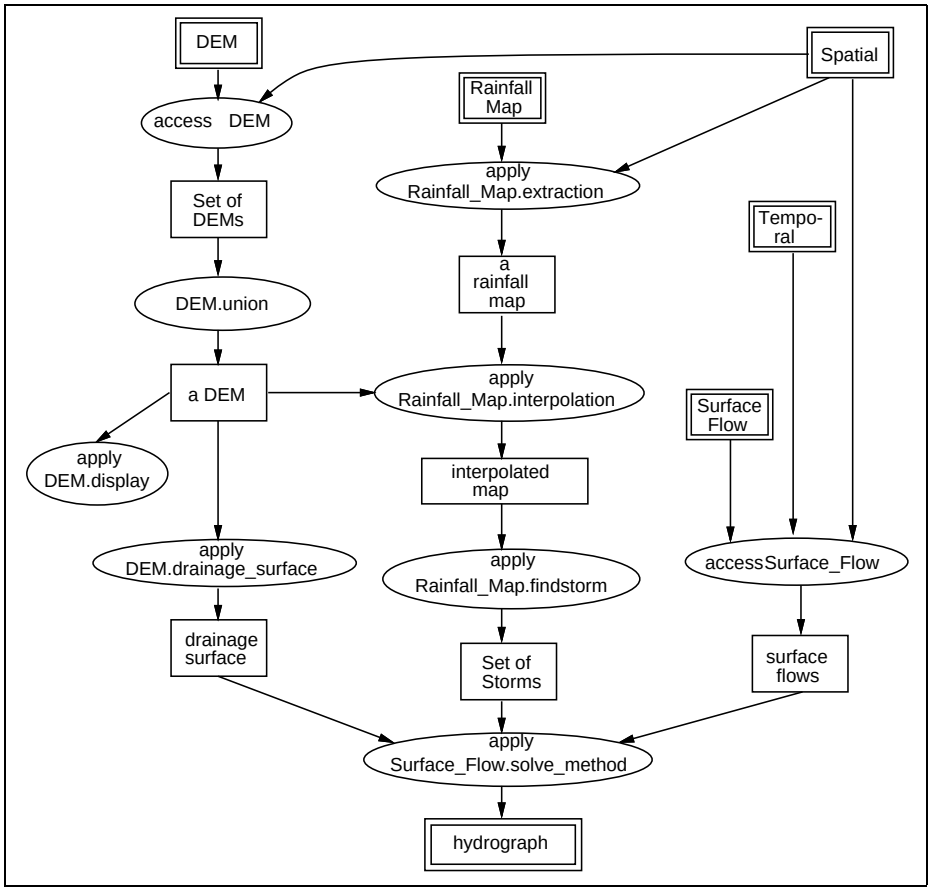


Figure 5: Modeling Schema **Surface-Flow-1** for Figure 1

of the tuple may be used to represent other aspects of a project such as the researchers of the project, the problems under study, and the duration of the project. An example of transformation in $\mathcal{T}$ of the project is *FindSubprojects* which gives a list names of all *sub-project* for specific given *project* name. Another example of transformation in $\mathcal{T}$ is called *FindParent* whose output is the *project* name for given a *sub-project*. The set of instances $\mathcal{I}$ of projects contains explicit representations of actual projects. One may view such R-structures as defining the access of groups of researchers to various subsets of the existing set of R-structures.

## 2.3 Computational Modeling Environments

We define a CME to be an integrated modeling environment that is based on the notion of R-structures. The basic operations that must be represented in a CME to permit scientists to construct R-structures for appropriate sets of modeling concepts include: (1) the definition, creation, manipulation and storage of new R-structures and their constituent parts; (2) the application of transformations to R-domain elements in general and to R-domain instances in particular;

and (3) the search for specific R-domain elements and transformations that satisfy appropriate constraints.

This basic functionality may be represented in terms of a simple *computational modeling language* (CML) [1] that possesses a small set of simple commands. This set includes the commands *create*, *delete*, *modify*, *access*, *store*, which may all be applied to R-structures, R-domains, transformations, and instances, as well as the command *apply*, which applies transformations to R-domain elements.

# 3 A Visual Computational Modeling Environment

Given the complex nature of modeling environments, expressive graphical representations of the support for model construction and testing is likely to be of great benefit to a scientist. In this section, we provide an overview of the architecture, functionality, and use of a VCME. The goal is to show how we may construct a relatively straightforward graphical interface that supports all the functionality of a CME based on our model of the scientific modeling activity. This interface involves two components. The first is a graphical environment for creating and manipulating projects, models, R-structures, and R-domains and transformations in an R-structure. The second component, which is described in Section 4, is a visual language for supporting the top-down construction of **modeling schemas::graphs**, an important class of R-structures that provide representations of our concept of models. This visual language may be viewed as a special but important component of the VCME that supports both the top-down and the bottom-up construction of models.

## 3.1 Functionality of the VCME

The VCME is based on our conceptualization of the scientific modeling process in terms of R-structures and transformations of R-structures. In our current design and implementation, the VCME takes the form of a graphical modeling environment, running under X-windows, in which users may create their own workspace and construct graphically the entities and transformations of an application using a toolbox of graphical symbols.

In the terminology of R-structures, a user has access to the graphical representations of operations for creating, registering, accessing, browsing, and searching R-structures and their various components (i.e., their R-domains, transformations, and instances of R-domain elements.). The user may also apply transformations to instances of R-domain elements. In particular, there is support for such operations on the special classes of R-structures that correspond to models and projects.

In terms of the more traditional terminology, the VCME provides access to the following functionalities:

1. **Project management**: A project in a CMS is viewed as an instance from the R-domain of a special R-structure for *Projects* (i.e., as a data element of a special type). The project manager provides support for the user to

create and register a specific project. The user can also search and browse through projects, and displays models related to a particular project.

2. **Model management**: A model in a CMS is viewed as an element from the R-domain of a special R-structure for modeling schemas. The representation of models comprises a general description, a set of input data domains (R-domains), an output data domain (R-domain), a set of functions (transformations), information characterizing, for example, the results of the model simulation, and a model ID. Basic modeling operations, such as $create$, $update$, $access$, and $model\text{-}execution$, are supported by the $model\text{-}editor$.

3. **Function management**: In a CMS, functions take the form of transformations between R-structure elements. These functions or transformations may be constructed graphically using a flowchart editor. This editor supports the definition of both "primitive" functions and functions defined using other functions, which we term "complex" functions.

4. **Data management**: In a CMS, data take the form of instances of R-structure elements. In general, such elements are best represented in terms of a complex data model, rather than a simple relational model. Since complex objects are created with the application of "constructors", such as *set, tuple, list,* and *sequence* constructors, the VCME provides graphical representations of these constructors, as well as supporting operator inheritance.

In addition to the above functionalities, the VCME also provides browsers that allow the user to visualize data, metadata, functions, models, and projects. The browsing activities that are supported include: the display of R-structures and their components, including R-domain elements and associated transformations; viewing both system-defined and user-defined meta-data; and visualizing the results of executing a model or, more generally, the results of applying a transformation to data domains.

## 3.2 A High-Level Computational Modeling Language (CML)

We describe here the simple, largely declarative language CML for expressing scientific modeling and database operations. Based on the concept of R-structures, CML is designed to express at the conceptual level, easily and naturally, most of the operations that are employed in iterative model development, while hiding irrelevant computational issues. This section gives a brief overview of some primitive language constructs.

The primary functionalities of CML include the definition, creation, manipulation and storage of new R-structures and their constituent parts; the application of transformations to R-domain elements in general and to R-domain instances in particular; and the search for transformations and specific R-domain elements that satisfy appropriate constraints. CML includes a small set of simple commands. The core CML commands are *create, delete, modify,*

*access* and *store* (R-structures, R-domains, transformations, relationships, and instances) and *apply* (transformations to R-domain elements).

The *create* command in CML permits the construction of abstract and concrete R-structures, and the components of such structures. In creating an R-structure, it is necessary to associate the appropriate R-domains, transformations relations and instances with the R-structure. It is possible to specify a set of *super* domains, that is R-structures from which this R-structure will inherit concrete representations, transformations and constraints. In particular, the R-domain of a super R-structure *contains* the R-domain of any sub R-structure. For example, DEM can be created by the following (where *peg* represents *point_elevation_grid*):

```
CREATE DEFAULT CONCRETE R-STRUCTURE DEM::peg
    SUPER R-STRUCTURES = {Rectangular_Grid_Maps::peg }
    R-DOMAIN = [name:string, resolution:integer,
                    location:[L1:point, L2:point, L3:point, L4:point],
                    P_E:set of [Location:point, Elevation:real]]
    CONSTRAINTS = · · ·
    TRANSFORMATIONS = {display_dem(DEM::peg):bool, · · · }
```

The user may also "name" the elements by values of type **string**. Although these names play the same role as object identifiers, this provision in CML provides flexibility and ease in scientific modeling activities. For example, if Y holds an identifier of a **DEM** element, the command "CREATE ELEMENT Manaus IN DEM VALUE = Y" creates a new name *Manaus* for the element. While each element in an R-structure can have 0, 1, or more user defined names, they must be unique in an R-structure and all its substructures, i.e., consistent with the inheritance hierarchy.

A key operation in CML is the application of transformations to elements from R-domains. Such applications may be expressed in CML in terms of the *apply* command. Suppose the variable Y holds a set of **DEM** element identifiers. The command "APPLY DEM.union TO Y" results in a (new) element of type **DEM**; it also returns the identifier of the new element which can be stored in another variable to be used later. The *apply* command has a large number of important applications which include the creation of *datasets* in an R-structure. The following example shows how we may create a new explicit instance of an R-domain element of the R-structure **DEM_SLOPES** using the transformation **DEM**.*compute_slope*:

```
CREATE ELEMENT IN DEM_SLOPES VALUE = APPLY DEM.compute_slope TO Y
```

The important but simple command *access* in CML allows queries on R-structures, their four main components and elements of their components. We illustrate with examples relating to the access of datasets. Assume that the abstract R-structure **DEM** has already been defined and that *Manaus* is a variable holding a spatial object identifier. Using the predicate *intersect* on pure spatial objects, we can find all DEMs whose spatial projection overlaps *Manaus*:

```
Y = ACCESS { X IN DEM
             WHERE DEM.spatial_projection(X) INTERSECT
                   spatial_projection(Manaus) }
```

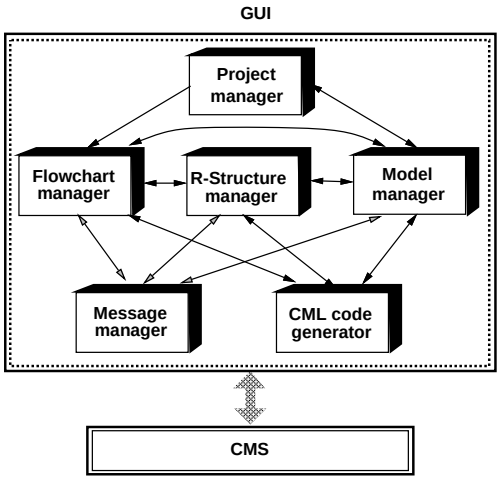


Figure 6: The VCME architecture

Local variables can be declared and used in the body and their scopes are within *begin–end*. New domains and elements can also be created with scopes within the function but they are not persistent unless explicitly saved (by the *store* command).

Further details of the language can be found in [2]. With the brief description of the language, we now discuss the organization of VCME.

## 3.3 Architecture and Windowing Systems of the VCME

The architecture of our VCME is depicted in Figure 6. It comprises seven components, each of which is represented graphically by a display window. The user interacts with the VCME through these graphical window systems to manage projects, models, functions, and data. These components, their associated display windows, and supported functionalities are:

1. A **project manager** through which the user may access, create, modify, and delete projects. The project manager also communicates with the model manager and the flowchart manager (described below) to update the metadata records and functions in its models.

   When a user first enters the VCME, the project manager creates a window in which all predefined projects are displayed (e.g., Figure 7). The project manager supports three types of operations: creating new projects, browsing existing projects, or invoking the flowchart editor. A new project is created by supplying the owner and application names. A short comment can be added to facilitate keyword search. The project manager registers

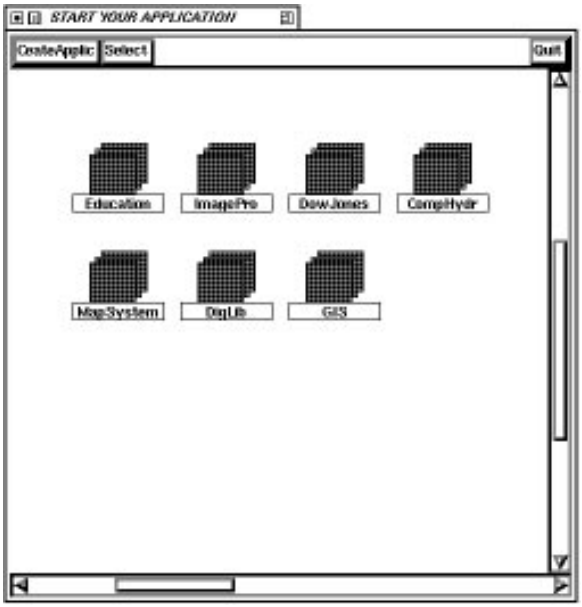


Figure 7: The project manager window

the new application project in the display window and saves the information in the system's metadata record.

The project manager allows the user to select an existing project by clicking on the project icon and creating a model management window. It invokes the model manager for displaying models within the selected project. The project manager also allows the user to access the flowchart manager which invokes the flowchart editor, the R-structure manager, the function library, and message display for editing a particular model in the selected project.

2. A **model manager** and its associated window provide a graphic environment for browsing, creating, and deleting models (Figure 8). The model manager also provides access to the data and procedures of a specific model by loading the flowchart work environment.

   There are two stages in the creation of an application model. The first stage is to create models in the model manager window by supplying the model name and a short comment for keyword search. This process only registers a model name and keywords in the system metadata records and causes the model to be linked with the associated project. This process, however, does not specify the computational procedures and R-structures used in the model, which means that the model is “empty”. In the second stage, the user invokes the flowchart editor, the R-structure manager, the function library, and the message display to expand the internal structure of the model (Figure 9).

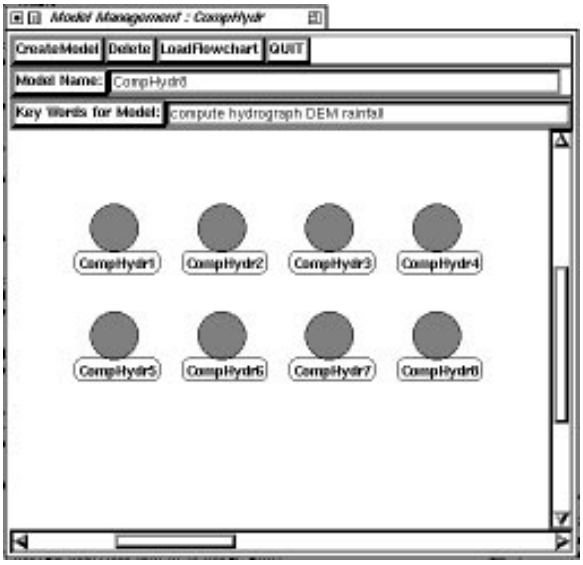


Figure 8: The model manager window

3. An **R-structure manager** is used to create, register, access, browse, and search R-structures and instances of R-domain elements. This R-structure manager window includes three sub-windows: one for abstract R-structures, another for concrete R-structures, and the third for instances of R-domain elements. Once the user selects a particular abstract R-structure, the concrete R-structure window displays all associated concrete R-structures. Similarly, when the user selects a concrete R-structure, the instance window displays all R-domain instances associated with the particular concrete R-structure.

   The user may create a concrete R-structure in the appropriate sub-window by supplying the concrete R-structure name and the name of its super R-structure (Figure 10). The domain of a concrete R-structure may be specified in three different ways: (1) with a domain name that has been previously defined; (2) with an external name defined by another package, e.g., MATLAB; and (3) using a CML script in which the user specifies the domain structure (Figure 10).

   In general the creation of abstract R-structures is automated. When the user creates a concrete R-structure, the system checks whether an associated abstract R-structure exists. If it does not, the system creates an appropriate abstract R-structure automatically. The user may also create an abstract R-structure manually by providing the abstract R-structure and the super R-structure names, and supplying the signatures of the associated transformations.

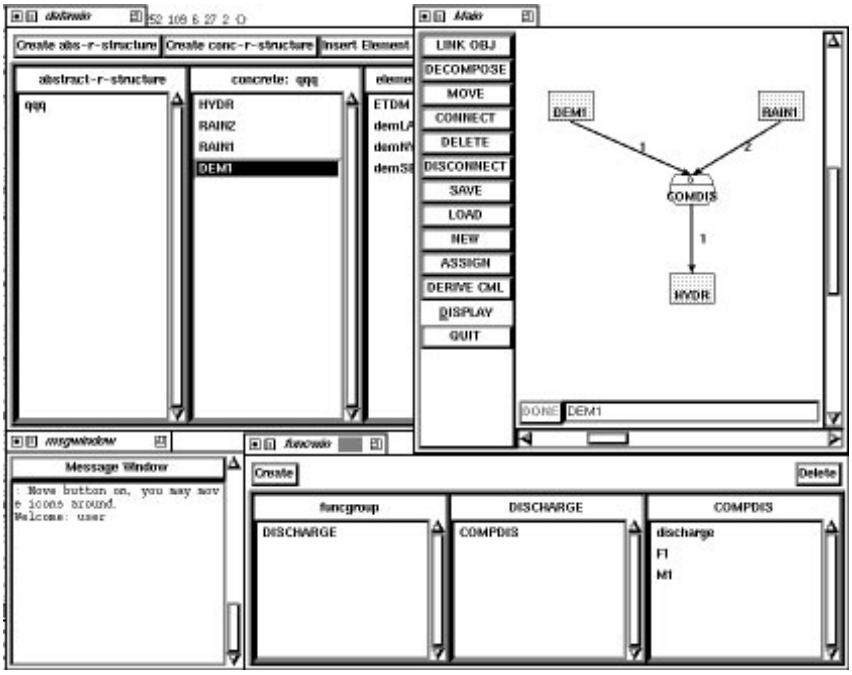


Figure 9: The VCME's model editing environment: the flowchart editor, the R-structure manager, the function library, and message display.

In the lower part of the workspace in Figure 10, the user may define the transformations associated with the concrete R-structure. The operation of saving an R-structure definition invokes a code generator (described below) that automatically derives a CML script encoding the data domains and transformations of the R-structure and saves the script in the DBMS (see Figure 11).

4. A **function-library manager** and its window which contains a catalog of functions (transformations) and models. The user can create new functions or register routines from other systems, such as Khoros or Grass. Another use of this window is to browse the hierarchy of the function library.

5. A **flowchart manager** whose window provides a "complete" graphical programming environment for the definition of transformations and models. On creating a transformation, the R-structure manager provides appropriate R-structures as inputs and outputs of the transformation. On creating a model, the model manager registers the model in a model base.

   When a flowchart is created by linking data domains and transformations, the flowchart manager performs checking on the number of parameters and the consistency of the data types. The flowchart manager creates the model schema and relays it to the CML code generator. It also communicates with the message manager for

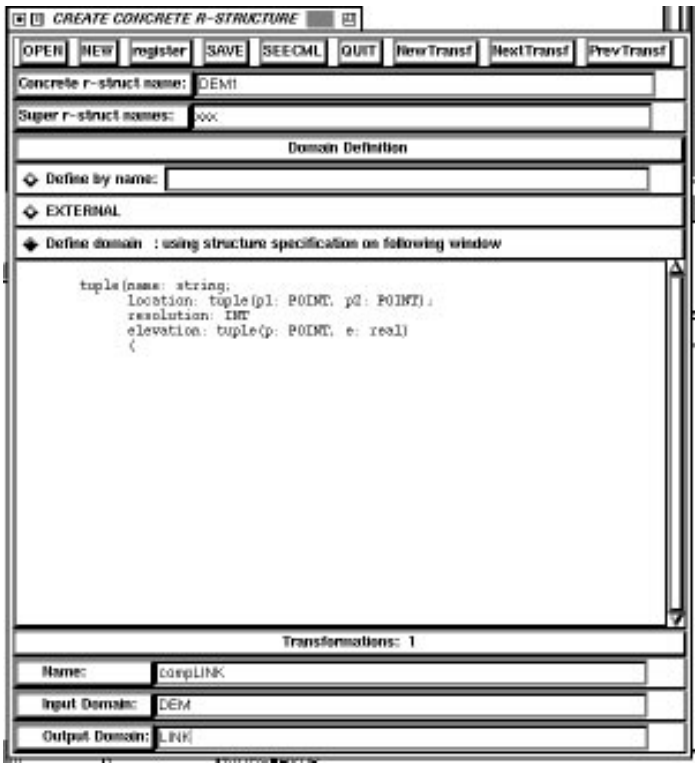


Figure 10: Creating a concrete R-structure.

displaying on-line help and error/warning messages.

6. A **message manager** and its window are used for on-line help and display messages generated within the VCME and from the CMS. For example, a message may be sent from the CMS to the GUI for error reporting.

7. A **code generation manager** and its window generate the CML script based on information sent from the R-structure, flowchart, and model managers. The code generator also sends the model schema to the CMS for storage.

Finally, a "**general manager**" coordinates the activities of the preceding seven managers withing the VCME, as well as communication with the environment outside of the VCME.

# 4 A Visual Language for Top-down Model Construction

A central feature of the VCME is support for both top-down and bottom-up model construction. As indicated in section 2, it is reasonable to view a large class of scientific models as directed graphs in which nodes are instances

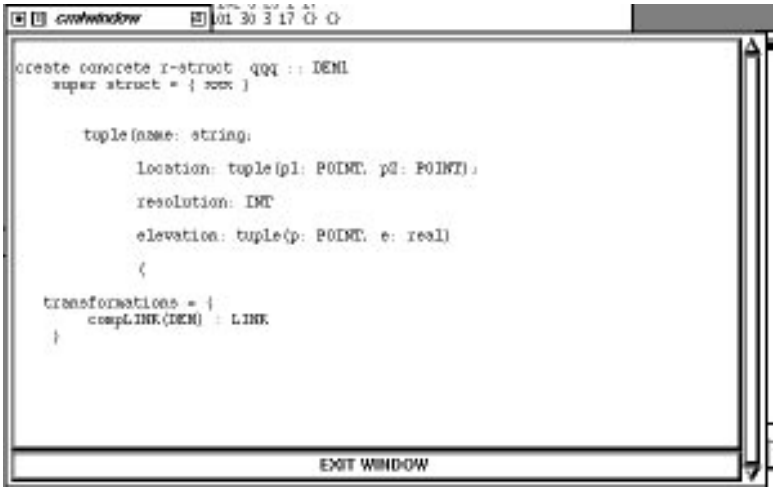


Figure 11: The CML code which encodes the concrete R-structure specified in Figure 10, generated automatically by the code generator

of R-domain elements or transformations, and arrows represent the directions of data flow. This idea is illustrated in Figure 5. There are clearly many ways in which scientists may create instances of models like the one in Figure 5. Typically, these involve various combinations of top-down and bottom-up constructions with iterative refinement.

Based on our interactions with applications scientists, we believe that top-down model construction is an especially important approach, representing in some sense the reductionist approach adopted by many scientists. In such an approach, higher-level phenomena and the associated modeling concepts are represented ("explained") in terms of lower-level phenomena and concepts.

In this section, we discuss support for the top-down construction of classes and instances of models in the VCME. In particular we describe a visual language that supports the top-down design of models represented as graphs. The top-down construction divides a coarse model into successively finer sub-models in which nodes are R-structures and transformations.

## 4.1 The Top-down Construction of Models

In a top-down modeling process, a scientist approaches a modeling problem by first specifying high-level inputs and outputs that are to be linked (explained) by some model. The detailed mechanisms linking the desired outputs to the specified inputs may not be initially understood and is typically represented by a black box. Successive refinements of the model are intended to define the details of the mechanisms in an iterative manner. During the process of making such details explicit, as noted above in the description of the hydrological modeling example, a scientist may well

find it necessary to employ new concepts. In the terms of our conceptual model of scientific activity, this requires the definition of new R-structures.

The goal of the visual language is to support the iterative refinement process by which an initial black box linking high-level inputs and outputs is decomposed into sequences of intermediate inputs and outputs and the associated transformations. The use of this visual language in VCME is supported by operations that include the construction of R-structures.

The flowchart editor of the VCME allows the user to define the input and output data domains, and to place an icon with a model name between these inputs and outputs for representing a decomposable transformation or model. The user then decomposes the icon and replaces it with R-domain elements, transformations, or other such decomposable icons, which flesh out the specific structure of the original icon or black box. Continuing in this manner, the user may create a sequence of decompositions, each representing a specific computation, until a level is reached at which the computation is fully specified in terms of existing R-domain elements and transformations. We call a decomposition level is terminated flowchart if it is fully specified. This means that any icon in this level represents an existing R-domain element, or an executable transformation coded by Fortran, C, C++, etc., or a predefined CML program. A terminated flowchart does not include any decomposed icon. During the iterative process of model specification, the user may at any time employ the functionality of the general VCME to construct any new R-structures or R-structure components that may be required by the modeling activity.

Such a process is shown diagrammatically in Figure 12, in which the first few steps of model development are illustrated. In the first-level decomposition, the user specifies input and output domains that are to be linked by the model *Hydrograph*. In this level the icon *Hydrograph* in flowchart is a black box. Successive refinements of the model are intended to define the details of the model *Hydrograph* in another flowchart aditor window. The process of deriving drainage surface from domain *DEM* is a new transformation which is still a black box whose detailed computation is specified in Figure 13. Similarly, the black box *Rainfall process* is specified in Figure 13. The transformations, *AccessSurfaceFlow* and *SurfaceFlowSolve* are existing Fortran programs that do not need continued decomposition. In particular, the combination of Figure 12 and Figure 13 leads to the relatively detailed data flow diagram illustrated in Figure 5. We may view Figure 5 as representing a fully specified model (assuming the existence of an appropriate set of R-structures and transformations).

## 4.2 An Illustration of Top-down Flowchart Construction

The flowchart manager employs layered window canvases for expressing each level of model decomposition: a decomposition icon (D-icon) is used to represent a decomposable model or transformation; a transformation icon (T-icon) is

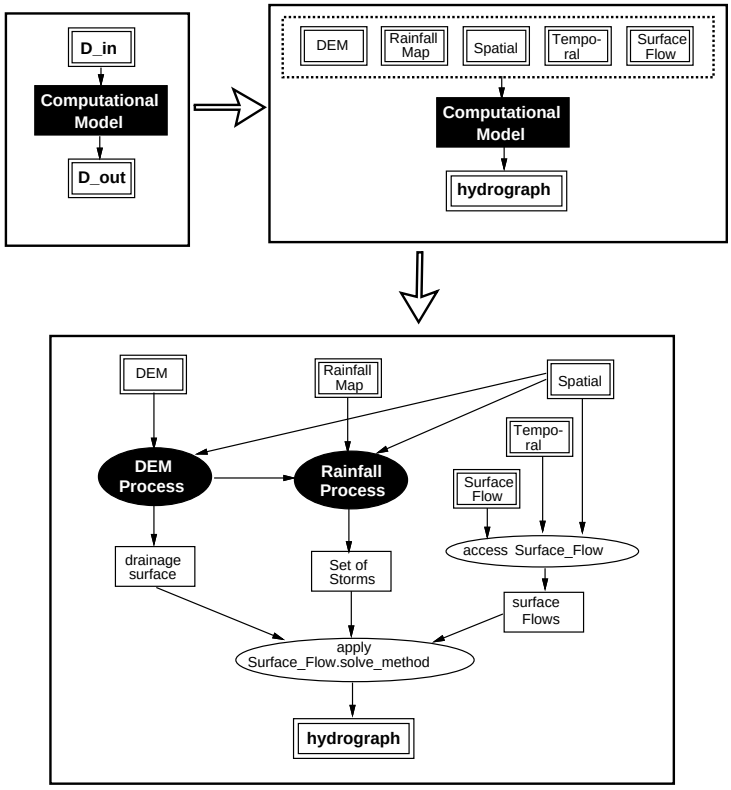


Figure 12: The first few steps of in the development of the hydrology model.

used to represent an executable function; and a data icon (Dt-icon) is used to represent input, output, or intermediate data domains.

Again the hydrological example may be used to illustrate how the visual language for top-down flowchart construction may be employed to support the construction and testing of models. Figures 14 and 15 show the initial and final snapshots of the flowchart editor window during a hierarchical, top-down model design process. We may view this process as occurring in five main steps.

STEP 1 *Specifying the model signature.* The signature of a model includes its input data, output data, and a D-icon labelled with the name of the model. In the hydrology example, input icons are created by dragging DEM and RAIN1 data from the concrete R-structure window to the flowchart canvas. Similarly, the output HYDR data (the hydrograph that results from applying the model to the input data) is created by dragging the concrete R-structure HYDR to the canvas. The D-icon linking the inputs and outputs is named COMDIS. The user then links the COMDIS icon to the model input and output icons, completing the construction of the initial flowchart

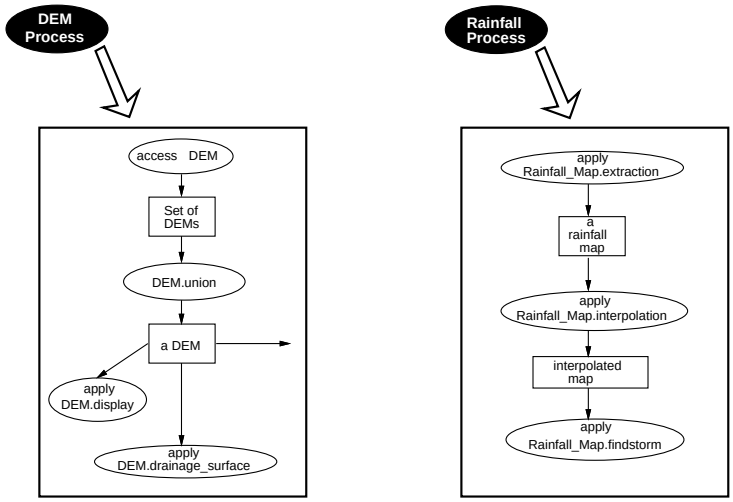


Figure 13: Decomposition of the hydrology model to a certain level of detail.

(see Figure 14.)

STEP 2 *Constructing the second level flowchart.* In the second step, the user expands the D-icon by double clicking on the COMDIS icon to create a new flowchart editor window. This editor is similar to the original flowchart editor but represents the first recursive step in the process of a top-down model design. In order to reduce ambiguity, the input R-structures in the new window are labelled DCINPUT1, DCINPUT2, etc. with reference to the decomposition icons in the original window. Hence in the current example, which involves the two inputs DEM and RAIN1, the inputs in the new window are labelled DCINPUT1 and DCINPUT2, as shown in Figure 15.

As illustrated in Figure 15, the user employs intermediate data domains and transformations to produce the model output. The intermediate data domain are “equal travel distance map” (ETDM) and “grid rain in area of Santa Barbara” (rainSB). The intermediate transformations involve the as yet undefined transformations COMdem, which maps the DEM data into the ETDM, and COMrain which maps RAIN1 data into rainSB. These transformations are initially represented as D-icons. The third transformation employed by the user, “discharge” is a predefined function that is an executable FORTRAN program registed in the function library. The icon for “discharge” is a T-icon created by dragging the discharge function from the function library and as such it cannot be further decomposed.

STEP 3 *Constructing the remainder of the flowchart.* The user repeats the previous steps until no more D-icon remain in any flowchart canvas. During this process the user may need to use the general functionality of the VCME

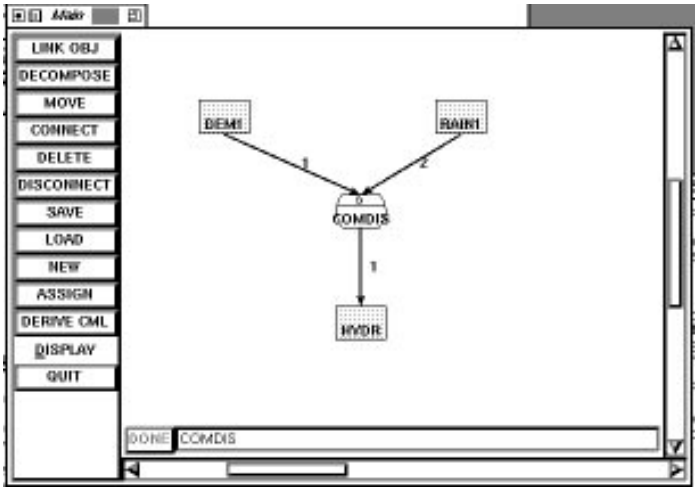


Figure 14: The flowchart editor window

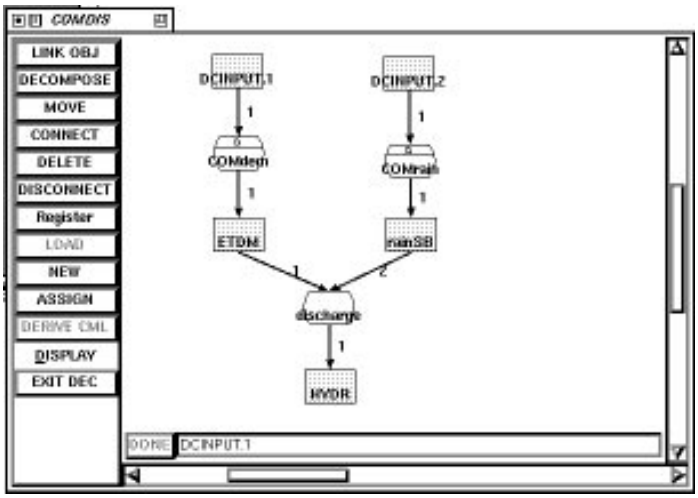


Figure 15: The flowchart editor window for icon “COMDIS” after decomposition

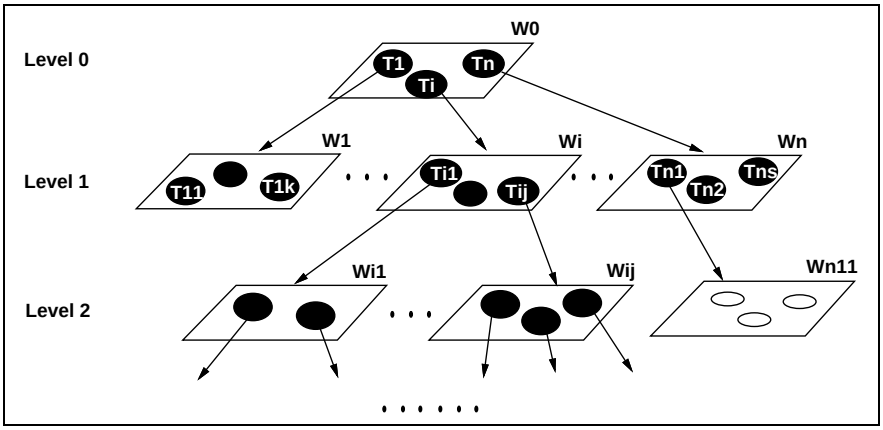


Figure 16: Top-down flowchart construction as a window tree

for creating appropriate R-structures. At this point, the process of initial model design is complete. Figure 16 illustrates the process of top-down flowchart construction. The windows created in the process form a tree where the head of the tree is the original flowchart window representing the top-level, abstract model description, while the tail of the tree is the concrete model with all data domains and transformations instantiated. In Figure 16, $W_0$ in $level\ 0$ of decomposition is the root window that includes D-icon $T_1,\ T_2,\ \ldots,\ T_n$. Window $W_i$ in $level\ 1$ represents D-icon $T_i$. $T_{i1},\ \ldots,T_{ij}$ in window $W_i$ all are D-icon which need to be defined by other flowcharts. A window is called a leaf if it does not contain any black box (D-icon). $W_{n11}$ in $level\ 2$, for example, is this kind of leaf window.

STEP 4 *Code generation.* Generating codes for a complex hierarchical flow diagram is in general a complex procedure. In particular, the number of levels of decomposition may be very large, and the graph representing such a flowchart may not be simple, since a node may represent another graph (or flowchart). We discuss briefly the main steps in the procedure for generating codes for the hydrological example. Before considering top-down code generation we describe a simple case of code generation processing for flowchart, without nesting decomposed icon or called case of non-hierarchical flowchart. We call simple process of code generation whose process can be expressed as follows:

(a) Type checking: in the event of linking data domain to transformation by drawing arrow, the code generator will perform the type checking operation.

(b) Link record: If the link from one icon to another is valid (only one case is invalid: link between two data domains), this link is recorded in a temporary place. Otherwise, an error message is sent to the message

window saying an invalid link was performed.

(c) Deriving code:

- Checking if defined flowchart is a connected graph; if not, send error message.
- Writing code: Starts from input domain, following the data-driven rule to derive the code, i.e. any function in flowchart will be called if its input data are ready.
- Continuing code generation until last icon is reached.

(d) Send CML code to DRAS.

For the complex case of nested hierarchical flowcharts:

- Starting at the root-level window $W_0$:

  Using above simple process of code generation, the generator develops the following *CML code*:

  ```
  HYDR = apply COMDIS to DEM, RAIN1
  ```

- After deriving full script for this level all D-icons will be considered, i.e., to consider next level:

  For all of D-icon, load the corresponding flowchart and derive script following the simple code process, for example, when the code generator identifies the D-icon “COMDIS”, it accesses the icon's decomposition window as indicated by a D-icon pointer and generates the following procedure:

  ```
  proc COMDIS (DCINPUT.1, DCINPUT.2, HYDR)
  {
    ETDM = apply COMdem to DCINPUT.1
    rainSB = apply COMrain to DCINPUT.2
    HYDR = apply discharge to ETDM, rainSB
  }
  ```

- If more D-icons remain, the above process is repeated.

In Figure 16, the main CML code is derived from the flowchart in window $W_0$ in $level\ 0$. The following is access to flowchart in window $W_1$ that specifies D-icon $T_1$. Then go to $W_2$ and so on. Then go to next level. The flowchart in window $W_{n11}$ does not contain any black box, so no continuing needed for this flowchart. The process will stop when all leaves window are reached.

The final body of codes is relayed to the DRAS. At model construction time, the real data elements of the R-structures are not yet defined. When running the model, the system prompts the user to instantiate the models by dragging appropriate elements of R-structures to the R-structure icons in the flowchart. In the preceding example, if the user drags the element "dem" to DEM1 and the element "rain" to RAIN1, and the model is registered as CompHygraph, then the code generator sends the string

```
run :: CompHygraph (dem, rain)
```

to the DRAS to cause execution of the model.

This concludes our discussion of the functionality and construction of the CME. A very important feature of the system is the support it provides for the access and manipulation of the data and computational programs (in C, Fortran - both stored as physical files), distributed on the network. We will now discuss how this support is provided in CMS and how it can be easily used through the CME.

# 5 Distributed Resource Access System in VCME

Until now, we have discussed how the modeling environment is organized and the process of modeling is supported in the CME. A key point to note here is that modeling scientific applications (an example is shown in Figure 1) requires a seamless integration of an extensible collection of external software tools and data stores, whose formats and behavior cannot be fixed a priori. This is particularly a challenge in the case of modeling activities that involve numerically-intensive and data-intensive applications. Not only should such support involve the integration and interoperability of data, diverse software packages, and distributed computing resources, but it should do so at a level of abstraction at which irrelevant computational details are hidden from the user. Thus, a CME **must** provide adequate support for such a distributed environment.

## 5.1 Requirements for Computation Support to CME

A first and clear requirement in this regard is to *bridge the gap between the high level modeling constructs and the low level details of system implementation*. Researchers faced with the task of using computers to develop and test complex models of phenomena typically work with a series of high level abstractions that are correctly interpreted only within the particular conceptual framework in which they are used. The construction, manipulation, and evaluation of models must therefore be described at a level such that the actual variables, routines, procedures, and even data sets used to represent the model are transparent with respect to the modeling tasks that are illustrated in Figure 1.

A second requirement is for *interoperability among the distributed data resources* that are used to support such modeling activities. The computational support used by members of scientific research teams is typically heterogeneous and distributed over a wide area network. It is clear that, in such an environment, scientists must copy (ftp) files to their local site and deal with the idiosyncrasies of the access method, host information, local storage capacity, and local file management. Also, data may have to be duplicated at several sites, introducing the problem of maintaining consistency of such data.

A third requirement is to provide a *uniform and generic interface to many different sets of tools*, such as software packages and computing devices. A user may require access to a large number of tools, such as compilers (C and FORTRAN), numerical packages (MatLab and Mathematica), geographical information systems (Arc Info, Grass), and image processing systems (Khorus). Dealing with these external tools is typically a difficult task because tool interoperability occurs at the granularity of a physical file. Users must modify locally-available programs and tools to address issues like site license restrictions for software, execution environments (path information, environment variables), and data format conversions across tools.

## 5.2 Resource Abstractions in DRAS

The DRAS has been designed to provide access to "external" data and tools in a distributed environment. In order to hide the heterogeniety of data and provide a consistent view, we have used R-structures to build a conceptual framework for an integrated computational environment. R-structures provide a high-level schematic view of physical datasets, and the functional aspects of external tools. Their instances have a representation that is "hidden" (from CMS), and hence are treated as black boxes and are managed as plain (binary or ASCII) files. In the example of Figure 14, data such as DEM and RAIN records are stored as follows. R-domain of external resources contain the following additional fields :

```
ACCESS METHOD : which can be one of FTP, Gopher, NFS, HTTP, CMS-RAP.
HOSTNAME      : Internet address of the machine for the resource
PORT NUMBER   : Port Number at the remote site for connection
FILENAME      : Host specific file name
TYPE          : Tool/ASCII/Binary/C/FORTRAN - different values
                depending on the context they are used.
```

Having specified the R-domains for the R-structures, we have two more key components in our model: *domain instances* and *transformations* on domain instances. Instances are typically the datasets like Rainfall and Soil, and tools

like FORTRAN executables containing some Erosion Model or interactive packages like Mathematica and MatLab. Typical transformations would be Read, Write, Invoke or Terminate.

The DRAS comprises two key components : the *Tool Management System* (TMS) and the *Data Access System* (DAS). In providing computational support, the diverse set of software tools used in the construction of models must be integrated into a coherent unit. Since most tools cannot be modified and it is not possible to make a priori assumptions about their behavior, such integration requires that communication with, and interfaces to, various tools be handled in a manner that is transparent to the user. Such support is provided by the TMS. By tools, we actually refer to any software package (like MatLab, Mathematica, Khorus) or an executable program that supports a standard-I/O interface (keyboard/terminal). Most of the legacy systems fall into this latter category. TMS allows users (interactive user or application programs) to execute, in one environment, scripts (or programs) for other software tools. This is done by executing the tools in the background while redirecting I/O. The design goal of the TMS was that it be built without altering the implementation of the tools since users typically do not have access to the source code; and without any specific assumptions about the set of tools to be supported, i.e., there should exist a capability for configuring new tools into the system dynamically at the users' request. Establishing interprocess communication with tools is a non-trivial task in UNIX. To overcome this, it uses the UNIX concept of pseudo-terminals which ensures that the integration of any software that supports a standard I/O interface can be integrated into the system with ease. Further details of the TMS can be found in [2].

The DAS forms the lowest level of the *Amazonia* architecture, as shown in Figure 17.

It deals with the idiosyncrasies of accessing information scattered across the network. Given the nature and characteristics of scientific data, any realistic approach to the problem of data access, precludes a centralized solution. Therefore, the DAS is built on a distributed architecture. It provides the following three basic features :

- a configurable and uniform interface to heterogeneous data access mechanisms
- support for data filtering and remote tool execution
- support for high-level data abstractions.

DAS is built on the client-server paradigm with a CMS site (DAS client) requesting data or tool access from a remote site (DAS server). The DAS uses its own new protocol, the CMS-Resource Access Protocol (CMS-RAP) to serve as its transport mechanism to transfer messages across participating sites on a network. CMS-RAP is built over the Hyper-Text Transfer Protocol (HTTP) which provides a clean interface to heterogeneous data access mechanisms (FTP, Gopher, WAIS) and is based on the client-server paradigm too. Also, the widespread use and easy installation of new HTTP daemons makes the DAS implementation open and scalable. For this reason, we have built the DAS

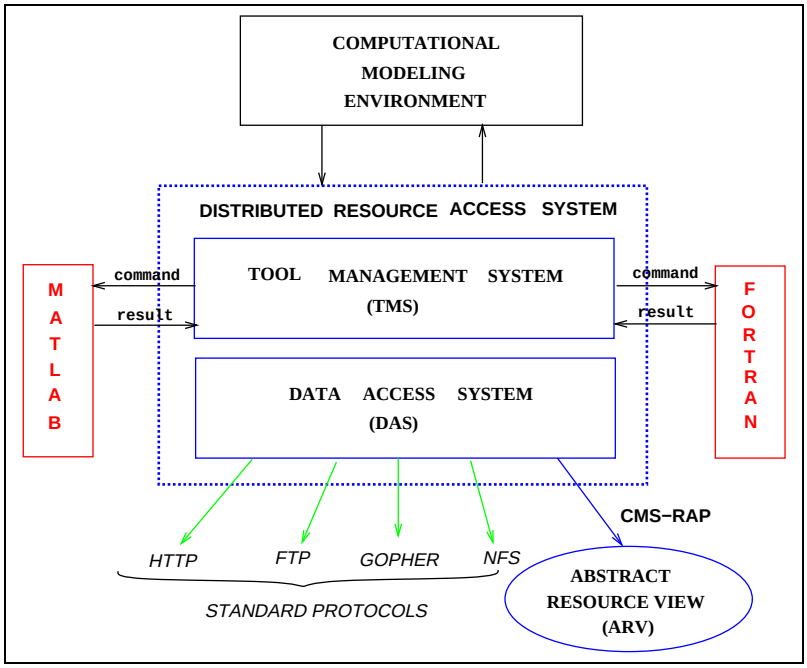


Figure 17: The architecture of DRAS

without resorting to the use of any peculiarities of particular HTTP daemons; or making modifications within the protocol itself.

## 5.3 Working Example

A very typical situation would be the following : a scientist at Santa Barbara (in the "cs.ucsb.edu" domain) wants to execute the Hydrology Model. The Rain Data needed as one of the inputs is available locally, but the data for Slope Length (for California) is stored by EOS scientists at *Seattle*. The output of the Hydrology Model is the Hydrograph data, which is viewed by the scientist in Santa Barbara using MatLab (located in the Math department). This is represented in Figure 18. Using CMS, such a complicated scenario is handled just by the specification of R-Structures.

```
NAME    : slope_length
ACCESS  : CMS-RAP (default)
HOST    : boto.earth.washington.edu
FILE    : /eos/DATA/california/SlopeLength
```

When the CMS executes the Hydrology Model, the system calls an "open" on the "slope_length" file. This invokes the DAS library, which dereferences the `slope_length`, as shown above. Subsequently, the CMS-RAP is used to

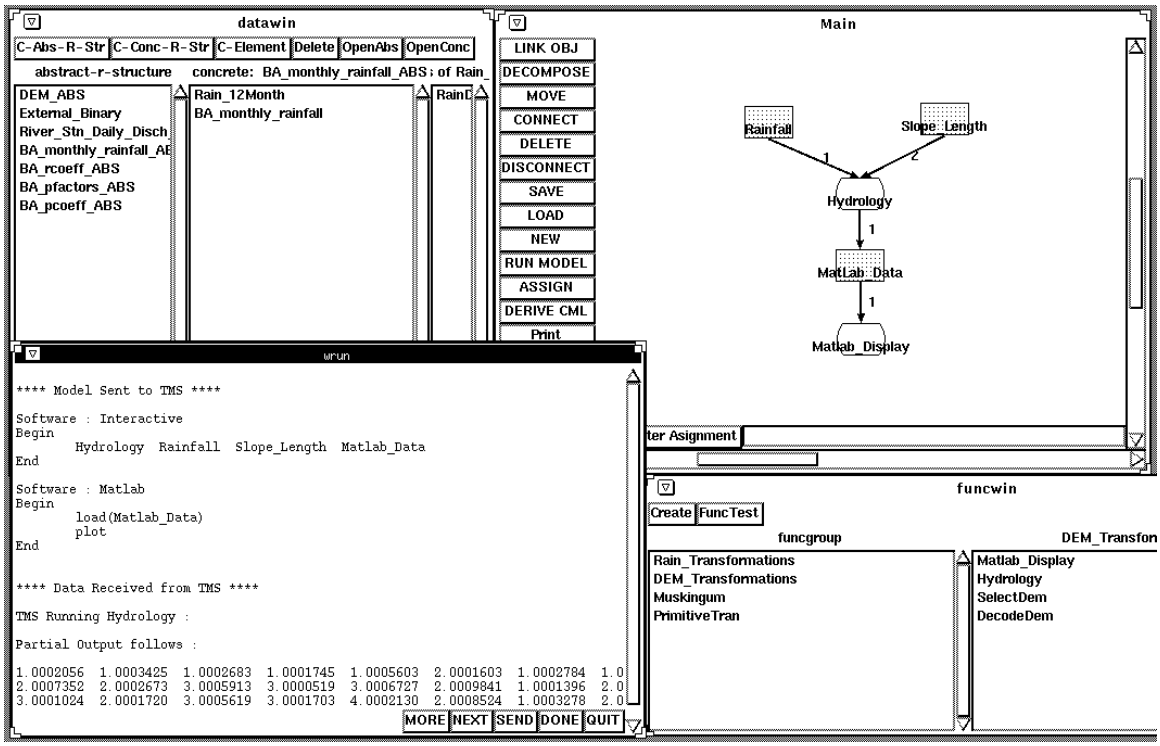


Figure 18: Interactive Execution of Models

communicate with the host `boto` in Seattle, Washington, the slope data for Santa Barbara is read, along with the rain data which is available locally. The output of the Model (hydrograph) are displayed interactively on a CMS window, as shown in Figure 18. The lower left window (window id : wrun) is the interactive window which pops-up when a model is run. It displays the Model/transformations sent to the TMS/DRAS for execution and the data received as output. The scientist has control over the operations to be performed, using the 5 buttons provided. The "MORE" button is used to display further data from the TMS which cannot all fit on the window. As the example illustrates, each transformation in the model is executed by the TMS sequentially. The "NEXT" button is used to initiate the execution of successive transformations (in our case, the execution of `Matlab` after the execution of the `Hydrology` Model. The "SEND" button is the most important feature of the VCME. This brings out the real interactive support for the scientist executing a model. The user can type inputs needed by the model during execution and "send" them to the model (waiting on the user for input) through the TMS. On receiving the input, the execution resumes and further data/inputs, if any, are displayed on the window. We have 2 kinds of ways in which the user can exit from an executing model. Using "DONE", the execution of the current transformation can be forced to end, automatically initiating the execution of the next transformation in the model. In our case, it is very useful to test the intermediate outputs of a

new transformation e.g., `Hydrology`. The user can click "DONE" midway, and initiate the execution of Matlab on the data output so far to determine the partial correctness of the result. This is typically helpful in computationally intensive applications that might complete in hours, where the user might like to test data during the initial runs. Finally, the "QUIT" button is used to end the execution of the complete model.

To display the hydrograph, the user needs to configure MatLab as follows :

```
NAME       : MatLab
ACCESS     : CMS-RAP (default)
HOST       : tool.math.ucsb.edu
FILE       : /usr/bin/math/matlab
TYPE       : Tool
```

Hydrograph is a concept for which the user has a R-structure defined, with several transformations associated with it. One of them for displaying data using MatLab, is as follows:

```
Display_Data(file)
{
    SOFTWARE : Matlab
    begin
       LOAD $file -ascii
       PLOT ($file)
    end
}
```

When the user executes `Display_Data(hydrograph)`, the transformation above is invoked. The first line `SOFTWARE` initiates a request to DRAS. The TMS subsequently dereferences MatLab, and sets a remote process at "tool.math.ucsb.edu" invoking MatLab. The commands (between the `begin` and `end`), are then passed interactively. The TMS considers these commands as character strings, and does not interpret them for semantic consistency. MatLab loads and displays the hydrograph which the DRAS displays at the user CMS window. The advantage of this approach is that the application developers can write many such small scripts and form a library. This library can be used by high-level clients directly without any knowledge of the underlying tools being invoked.

# 6 The Implementation and Use of the VCME

In this section, we describe the key elements in the implementation of a VCME on SUN workstations running the UNIX Operating System under the X-Window environment. The implementation employed the Tcl/Tk Toolkit and the C programming language. This version of the VCME provides graphical display of the model design process and the interactive manipulation of complex objects and computational flowcharts. It employs metadata records for supporting the GUI, and supports automated CML code translation from the GUI specification. We also indicate how a user interacts with the VCME.

## 6.1 Database Support for VCME

R-structures provide a language for constructing and manipulating representations of concepts and are the foundation on which the modeling environment of a CMS is constructed. Due to the hierarchical nature of the relationships between R-structures and the association of transformations with specific R-structures, the CMS data model lends itself to a convenient implementation based on object-oriented approaches following the ODMG specifications [41]. Object-oriented technology has been used in the development of techniques for integration of heterogeneous data management systems. In particular, [40] shows that the object paradigm not only solves the integration problems but also extends its scope. Furthermore, the inheritance mechanism of the object-oriented paradigm provides flexibility in accommodating tailored views of underlying applications.

Based on the object-oriented framework, R-structures of VCME are mapped onto **classes** which preserve the relationship between the R-structures. Further, R-domains are mapped to ODMG **types**. Primitive data types such as INTEGERs, REALs, and CHARACTERs are defined in ODMG. Complex types involving the set, tuple and sequence constructors are mapped onto Set, Structure, and List respectively of ODMG. Transformations defined on R-structures are mapped onto **methods** defined for the corresponding classes. The application of a transformation on an R-structure instance is analogous to the invocation of a method on the corresponding object. Instances of R-structures map onto **objects** belonging to classes. In implementing VCME we used the OODBMS $O_2$ [39] because it supports ODMG 93. We have taken particular care to ensure that the functionality of the CMS is not limited to that provided by $O_2$. For this, we have developed an interface providing high-level functions to manipulate $O_2$ objects. This interface is built primarily on ODMG specifications and is sufficiently generic to accommodate any standard OODBMS. We will discuss how $O_2$ is used in the following sections.

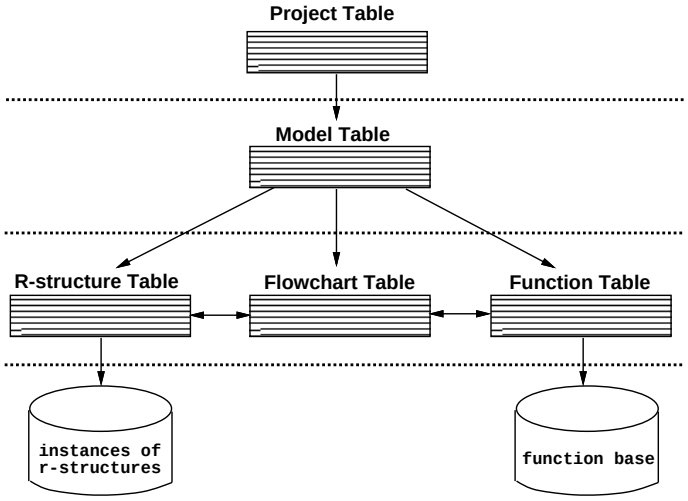


Figure 19: VCME's metadata structure

## 6.2 Metadata of the VCME

The five important management entities: projects, models, flowcharts, function libraries, and R-structures are recorded in a hierarchical metadata structure as depicted in Figure 19. The implementation employs complex tables in which an entry can be a simple data item, a pointer, or a set of data, to store the system metadata (vs. the simple table used in relational database where entries are simple data elements). Five such tables are used in our VCME: an R-structure table, a function table, a flowchart table, a model table, and a project table. These tables are stored in the $O_2$ database system. At run time the VCME general manager loads the metadata records from $O_2$ and maintains them in the local file system. Updates to the metadata records become persistent when they are restored in $O_2$ at the end of an interactive session.

Figure 20 illustrates the system metadata tables for the hydrological modeling example given in section 2. In Figure 20, the project table contains a list of application domains. When the system first starts up, the project window pops up and a set of project icons, corresponding to the application domains in the project table, await the user's selection. If the user clicks the “Hydrology” icon, the GUI manager will read in the model table associated with the hydrology project and display the hydrology application models in the model window. In Figure 20, for example, there are two such models, *channel* and *rasters*, for the hydrology drainage basin computation.

The structures of the computation procedures carried out in these models are depicted as flowcharts, which are stored in the flowchart table and linked with the particular models by pointers as depicted in Figure 20. When the user clicks on the icon “model-1” in the model window, the flowchart manager will parse the flowchart table and display

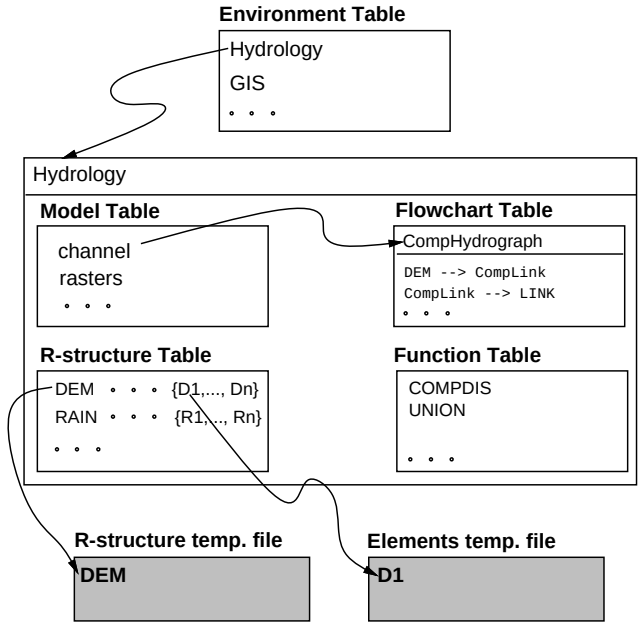


Figure 20: The system management tables in the VCME system

the flowchart in the flowchart editor canvas. The detailed description of the data and functions used in the computation are found in the R-structure and function tables, respectively. The real data sets, e.g., the DEM and RAIN data sets used in the hydrology example, are stored in UNIX files, which are accessed at run time by the CMS kernel.

A model is represented graphically as a flowchart. A flowchart comprises a set of objects that can be a data domain, a transformation, or (recursively) another model flowchart. The connections of a data domain to a transformation or a transformation to a data domain are also considered as objects. In a flowchart this connection is depicted as an arrow with the direction indicating the data flow. A model table comprises records of all the models associated with a particular project, including the model name, input domains, output domains, user comments, and flowchart pointers. For example, the model table for the hydrology project is shown in Figure 21. The model table is automatically generated by the model manager, except for the model name and user comment fields, by parsing the flowchart table and extracting the input and output domain names and the flowchart pointers.

A flowchart is drawn by the user with the aid of the flowchart editor. The system manager creates a flowchart table for manipulating and displaying flowcharts. Furthermore, the CML script encapsulating the functionalities of a flowchart is automatically derived and stored in the flowchart table. As shown in Figure 22, which depicts the flowchart table of the hydrology project, a flowchart table contains the window IDs (a root or a decomposition window), the object names (a data domain icon or a transformation icon), the object states (a data domain or a transformation, or

**Model Table**

| model name | input domain | output domain | comment | flowchart pointer |
|---|---|---|---|---|
| CompDisc-1 | DEM, RAIN | HYDROGRAPH | Hydrology application using algorithm-1 | Flowchart-1 |
| CompDisc-2 | CHANNEL RAIN | HYDROGRAPH | Using algorithm-2 | Flowchart-2 |
| . . . | | | | |

Figure 21: The model table in VCME system

**Flowchart Table**

| obj name | status | windowID | location | connectto | connectfrom |
|---|---|---|---|---|---|
| DEM1 | data | .0 | 90, 74 | COMPDIS | ϕ |
| RAIN1 | data | .0 | 298, 74 | COMPDIS | ϕ |
| COMPDIS | decomp | .0 | 210, 157 | HYDR | DEM1 RAIN1 |
| COMdem | decomp | .0.5 | 102, 102 | ETDM | DEM1 |
| COMrain | decomp | .0.5 | 252, 109 | RAIN1 | rainSB |
| ETDM | data | .0.5 | 102, 173 | discharge | COMdem |
| rainSB | data | .0.5 | 253, 173 | discharge | COMrain |
| discharge | transf | .0.5 | 181, 237 | HYDR | ETDM rainSB |
| HYDR | data | .0.5 | 182, 306 | ϕ | discharge |

Figure 22: The flowchart table with data examples

a decomposition icon), the locations of icons in the flowchart, the input icon connection lists, and the output icon connection lists.

## 6.3 Code Generation

Automated code generation is a desirable feature of a CMS because it improves user productivity, promotes code reuse, facilitates documentation, and provides a means of rapid algorithm prototyping [20]. We have found, in line with the claims of Rich and Waters [26], that most of the benefits of automatic code generation may be traced to code reuse and ease of maintenance.

The GUI-driven code generation process of our VCME is shown in Figure 23. The system metadata records with R-structures, transformations, library functions, and pre-defined models form the basis of the generator. Supplemental user information and the flowchart derived from the flowchart editor are inputs to the generator to produce automatically a CML script which encapsulates the functionalities of the flowchart. Three types of codes are produced by the code generator. The first type is code generated using the textual information supplied by the user; an example being the codes for the creation of a concrete R-structure using the R-structure name, its super R-structure name, its domain specification, and associated transformations.

The second type involves the CML scripts that are generated in response to a graphical flowchart specification

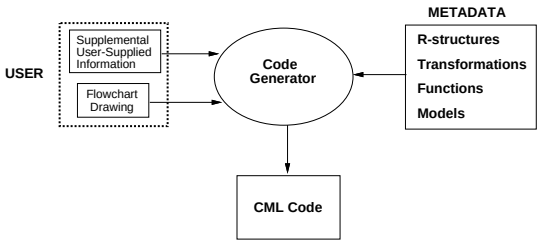


Figure 23: An overview of the code generation process

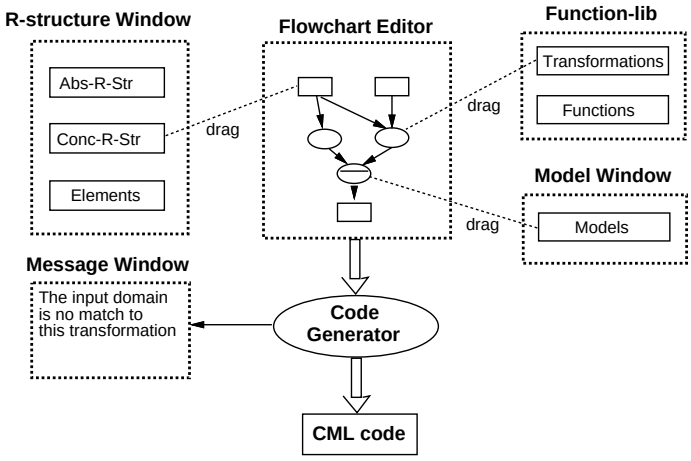


Figure 24: The processing of code generation for a flowchart

[1]. The third type involves control commands and special text strings for use by the Tool Management System of the DRAS and other tools. Examples include the generation of source codes in the form of UNIX commands, such as for the invocation of an external tool like Arc/Info or the initiation of communications with remote servers, and special text strings for use by the Tool Management System, such as a network address of an external tool or dataset.

Code generation from a flowchart is illustrated in Figure 24. In this process, the flowchart editor controls the drawing of a flowchart. The user drags-and-drops R-structures from the R-structure display window to the flowchart editor canvas for specifying the input, output, and intermediate data domains of a model. The user can also instantiate library functions and transformations from the library function window and reuses predefined models. Once a link is created between a data domain and a function, the code generator performs type consistency checking. The type information is obtained from the model table for the particular R-structure and the transformation signatures from the

[1] The current implementation still requires additional verification and validation algorithms for type checking, and drawing consistency checking.

function table.

When the code generator receives the command “DERIVE CML” from the flowchart editor, the generator parses the flowchart to generate the corresponding CML script and relays the script to DRAS. Starting from the top-level flowchart, the process of generating CML scripts comprises the following steps:

1. *Graph structure validation*: If the flowchart is made of multiple pieces, an error message is sent to the message window and the process stops, otherwise, it continues to next step.

2. *Topological sort*: The code generator consults the edge records stored in the flowchart table and derives a linear ordering of transformations in the flowchart.

3. *Derivation of CML script*: Starting with the first transformation in the linear order, to the last transformation, use the CML *apply* construct to execute the particular transformation to the input data domains to generate the output domain. When a D-icon is reached, the code generator accesses the icon's decomposition window as indicated by the D-icon pointer and recursively generates the CML script for this decomposable procedure.

4. *Registration of CML scripts in CMS*: After the CML script for the last transformation is derived, the code generator sends the whole CML script to DRAS. In our current system, we implement VCME and DRAS communication using a client-server model, so that VCME and DRAS can run on different hosts in a distributed environment.

Since the elements of R-structures are not instantiated at the time of flow-chart editing, the input data to the flowcharts are only generic R-structure names. Before executing a model, the system prompts the user to instantiate input data domains by dragging an element of the particular R-structure to the R-structure icon in the flowchart. After data instantiation, the code generator sends a string containing the data file name to DRAS for retrieving the data file. The code generator also specifies a file name for the VCME to store the returned result.

# 7 Concluding Remarks

The most important contribution of this paper is a description of the design and implementation of a *visual computational modeling environment* (VCME). The VCME essentially provides the graphical user interface to the full functionality of a *computational modeling system* (CMS). A CMS [1] supports the construction and testing of symbolic models of a broad range of phenomena. In particular, the VCME is a translation of an integrated computational modeling environment (CME) that is based on a comprehensive conceptualization of the scientific modeling process underlying a CMS. This conceptualization employs the concept of representational structures (R-structures).

The VCME permits a scientist to carry out, at an appropriate level of abstraction, any operation involved in the process of symbolic model construction, testing, and application. In general, the VCME permits a scientist to construct and manipulate, easily and with the use of graphical tools, representations for any modeling concept that might be required, as well as explicit instances of such representations and the transformations that operate on these instances. In particular, the VCME incorporates a new visual language whose use greatly facilitates the top-down (as well as the bottom-up) construction of scientific models. Such models are represented in terms of a special class of R-structures called *modeling schemas*, which in turn are constructed from the components of an iteratively-constructed set of R-structures representing appropriate scientific modeling concepts.

Both the design and implementation of the CMS/VCME have involved close collaborations with several environmental science research groups. We are currently collaborating with these groups on the implementation of full-scale modeling applications from the environmental sciences in terms of our current version of a CMS/VCME. Our experience to date, although limited, is that the use of a CMS/VCME greatly facilitates the iterative and complex process of constructing scientific models.

# References


[1] T.R. Smith, J. Su, A.E. Abbadi, D. Agrawal, G. Alonso and A. Saran (1995) Computational Modeling System, *Information System*, Vol. 20, No. 2, 127-153.

[2] T.R. Smith, J. Su, and A. Saran (1994) Virtual Structures – A Technique for Supporting Scientific Database Applications, 13th Int'l. Conf. on ER Approach, UK, *Lecture Notes in Computer Science*, Vol. 881.

[3] N.C. Shu (1988) Visual Programming, *Van Nostrand Reinhold Com.*, New York, 14-15.

[4] S.K. Chang (1987) Icon Semantics: A Formal Approach to Icon System Design, *International Journal of Pattern Recognition and Artificial Intelligence*, Vol. 1. No. 1. 103-120, Apr.

[5] S.K. Chang, J. Reuss and B. H. McCormick (1978) Design Considerations of a Pictorial Database System, *International Journal on Policy Analysis and Information Systems*, Vol.1, No. 2, 49-70.

[6] N. Roussopoulos and D. Leifker (1984) An Introduction to PSQL: A Pictorial Structured Query Language, *Proceeding of the 1984 IEEE Computer Society Workshop on Visual Languages*, Hiroshima, Japan, 77-87.

[7] C.F. Herot (1980) Spatial Management of Data, *ACM Transaction on Database System*, Vol. 5, No. 4, 493-514.

[8] M. Hirakawa, N. Y. Monden, M. Tanaka and T. Ichikawa (1986) HI-VISUAL: A Language Supporting Visual Interaction in Programming, *Visual Language*, edited by S.K. Chang et al., Plenum Presh, 233-259.

[9] National Instruments Corporation (1987) LabVIEW: a demonstratin, *National Instruments Corp.*, 12109 Technology Blvd., Austin, Texas 78727-6204.

[10] S. Matwin and T. Pietrzykowski (1985) PROGRAPH: A preliminery report, *Computer Languages* 10. 91-126.

[11] J. Rasure, et al. (1988) XVision: A Comprehensive Software System For Image Processing Research, Education and Applications, *ACM SIGGRAPH User Interface Software Symposium*, Banff, Alberta, Canada.

[12] J. Gold (1989) Digital and Imaging, *Advanced Imaging*, VV, 24-28.

[13] C.G. Masi (1989) Imaging with Icons, *Test and Measurement Woild*, VV, 85.

[14] A. Wilson (1989) A Picture's Worth a Thousand Lines of Code, *Electronic System Design*, VV, 56-60.

[15] C. Upson, et al. (1989) The Application Visualization System, *IEEE Computer Graphics and Applications.*

[16] B. Lucas, et al (1992) An Architecture for a Scientific Visualization System, *Proc. 1992 IEEE Visualization Conference*, Boston, MA.

[17] IBM (1995) IMB Visualization Data Explorer QuickStart Guide, *Second Edition*.

[18] IBM (1995) IMB Visualization Data Explorer User's Guide, *Six Edition*.

[19] M. Stonebraker, et al (1993) Tioga: Providing Data Management for Scientific Visualization Applications, *Proc. 1993 VLDB Conference*, Dublin, Ireland.

[20] J. Rasure, D. Argiro, T. Sauer, and C. Williams (1990) Visual Language and Software Development Environment for Image Processing, *International Journal of Imaging Systems and Technology*, Vol.2.183-199.

[21] A. Wolf (1990), How to Fit Geo-Objects into Databases — An Extensibility Approach, *Proc. of the First European Conference on GIS*, Amsterdam.

[22] W. Waterfeld and H.-J. Schek (1992), The DASDBS Geo-Kernel - An Extensible Database system for GIS, *Three Dimensional Modeling with Geoscientific Information Systems*, Kluwer Academic Publishers, Ed. A. K. Turner, 69-84, Netherlands.

[23] A. Segev (1993), Processing Heterogeneous Data in Scientific Databases, *Proceedings of the NSF Scientific Database Projects, AAAS Workshop on Advances in Data management for the Scientist and Engineer*, Ed. W. Chu, Boston, MA.

[24] C. B. Medeiros and F. Pires (1990), Databases for GIS, *SIGMOD Record*, Vol.23, No. 1, 107-115.

[25] C. V. Jones (1990), An Introduction to Graph-Based Modeling Systems, Part I: Overview, *ORSA Journal on Computing*, Vol.2, No. 2, 136-151.

[26] C. Rich and R.C. Waters (1988) Automatic Programming: Myths and Prospects, *IEEE Computer*, VV. 40-50.

[27] C. M. Bowman, P. B. Danzig, D. R. Hardy, U. Manber and M. F. Schwartz (1994) The Harvest Information Discovery and Access System, *Proc. 2nd Int. WWW Conf.*, Chicago.

[28] Shelley G. Ford and Robert C. Stern (1994) OmniPort: Integrating Legacy Data into the Web, *Proc. 2nd Int. WWW Conf.*, Chicago.

[29] L. Shklar, H. Marcus, A. Sheth and S. Thatte (1994) The "InfoHarness" System for Integrated Information Access and Management, *Proc. 2nd Int. WWW Conf.*, Chicago.

[30] G. Mathews and S. S. Towheed (1995) NSSDC OmniWeb: The First Space Physics WWW-Based Data Browsing and Retrieval System, *Proc. 3rd Int. WWW Conf.*, Germany.

[31] K.J. Maly et al (1995) Mosaic + XTV = CoReview, *Proc. 3rd Int. WWW Conf.*, Germany.

[32] E. Mesrobian, R. R. Muntz, J. R. Santos, E. C. Shek, C. R. Mechoso, J. D. Farrara and P. Stolorz (1994) Extracting spatio-temporal patterns from geoscience datasets, *Proceeding IEEE Workshop on Visualization and Machine Vision*, June, 92-103, Seattle.

[33] F. Long and E. Morris (1993) An Overview of PCTE: A Basis for a Common Tool Environment, *Tech. Report Carnegie Mellon University, Pittsburg*, CMU/SEI-93-TR-01-ESC-TR-93-175.

[34] Y. Ioannidis, M. Livny, E. Haber, R. Miller, O. Tsatalos and J. Wiener (1993) Desktop Experiment Management, *IEEE Bulletin of the TC on Data Engineering*, Vol.16, No. 1.

[35] J. B. Cushing, David Hansen, David Maier and Calton Pu (1993) Connecting Scientific Programs and Data Using Object Databases, *IEEE Bulletin of the TC on Data Engineering*, Vol.16, No. 1.

[36] R. G. G. Cattell et al (1994) The Object Database Standard ODMG-93: Release 1.1, *Morgan Kaufmann*.

[37] Object Management Group (1992) The Common Object Request Broker: Architecture and Specifications - 91.12.1 Revision 1.1, *OMG Document*.

[38] M. J. Carey, and D. J. DeWitt, M.J. Franklin, N.E. Hall, M. McAuliffe, J.F. Naughton, D.T. Schuh, M.H. Solomon, C.K. Tan, O. Tsatalos, S. White and M.J. Zwilling (1994) Shoring Up Persistent Applications, *Proc. ACM SIGMOD*, 383-394.

[39] C. Lecluse, P. Richard and F. Velez (1988) $O_2$: An Object-Oriented Data Model, *Proc. ACM SIGMOD*, Chicago. 424-433.

[40] E. Bertino, M.Negri, G. Pelagatti and L. Sbattella (1994) Applications of Object-Oriented Technology to the Integration of Heterogeneous Database Systems, *Distributed and Parallel Databases*, Vol.2, No. 4, 343-370.

[41] Keith A. Carlson (1994) Use of Object-Oriented Constructs in a Computational Modeling System for Earth Scientists, *Masters Thesis, Computer Science Department, University of California, Santa Barbara.*