Programming Environments for Multidisciplinary Grid Communities

Naren Ramakrishnan, Layne T. Watson, Dennis G. Kafura,
Calvin J. Ribbens, and Clifford A. Shaffer
Department of Computer Science
Virginia Tech, Blacksburg, VA 24061

July 20, 2001

Abstract

Rapid advances in technological infrastructure as well as the emphasis on application support systems have signaled the maturity of grid computing. Today's grid computing environments (GCEs) extend the notion of a programming environment beyond the compile-schedule-execute paradigm to include functionality such as networked access, information services, data management, and collaborative application composition. In this article, we present GCEs in the context of supporting multidisciplinary communities of scientists and engineers. We present a high-level design framework for building GCEs and a space of characteristics that help identify requirements for GCEs for multidisciplinary communities. By describing integrated systems for five different multidisciplinary communities, we outline the unique responsibility (and opportunity) for GCEs to exploit the larger context of the scientific or engineering application, defined by the ongoing activities of the pertinent community. Finally, we describe several core systems support technologies that we have developed to support multidisciplinary GCE applications.

Contents

1 Introduction

Grid computing environments (GCEs) have matured significantly in the past few years. Advances in technological infrastructure as well as a better awareness of the needs of application scientists and engineers have been the primary motivating factors. In particular, the shift in emphasis from low-level application scheduling and execution to high-level problem-solving [16] signals that grid computing will become increasingly important as a way of doing science. We use the term GCE to broadly denote any facility by which a scientist or engineer utilizes grid services and resources to solve computational problems. Our definition thus includes facilities from high-performance scientific software libraries [55] augmented with grid access primitives to domain-specific problem-solving environments (PSEs) [24, 71] that provide targeted access to applications software.

GCEs extend the notion of a programming environment beyond the compile-schedule-execute paradigm to include functionality such as networked access [15], information services, data management, and collaborative application composition. This is especially true when designing such systems for supporting **multidisciplinary grid communities**, the focus of this paper. Our emphasis at Virginia Tech has been on exploiting the high-level problem-solving context of such 'virtual organizations' [35] and building on the grid architectures, services, and toolkits (e.g., [8, 33, 79, 77]) being developed by the grid community. In working with large, concerted groups of scientists and engineers in various applications (aircraft design, watershed assessment, wireless communications system design, to name a few), we have identified several recurring themes important for supporting and sustaining such communities. Our goal in this paper is to document these themes, present a high-level design framework for building GCEs [36], and describe some solutions we are working on to address the concomitant needs.

In the remainder of this section, we present usage scenarios from multidisciplinary communities that will help us characterize requirements for GCEs. We also describe a high-level framework for building and organizing programming environments for such communities. In Section 2, we describe PSEs that we have built for five different multidisciplinary communities. Section 3 discusses a variety of issues pertaining to software systems support for GCEs. In particular our semistructured data management facility plays a central role in exploiting the rich problem-solving context of multidisciplinary grid communities. Two other elements of our GCE framework are described in Section 3: Sieve (a collaborative component composition workspace) and Symphony (a framework for managing remote legacy resources). We conclude with a brief discussion of future directions in Section 4.

1.1 Multidisciplinary Grid Communities: Scenarios

We begin by describing some scenarios to illustrate the needs typical of multidisciplinary grid communities. We posit that there are fundamental differences in the usage patterns for a single researcher (or even a group of collaborators) working on a relatively homogeneous problem as compared to the usage patterns found in the communities we have in mind. For example, how does a grid community for solving matrix eigenvalue problems differ from, say, one for aircraft design or wireless communications? We identify three scenarios that are suggestive of the distinctions we would like to make.

- *Scenario 1:* A specialist in ray tracing, a channel modeler, and a computer scientist are addressing the problem of determining the placement of wireless base stations in a square mile area of a large city such that the coverage is optimal [51]. In terms of execution, this problem involves a computation expressed as a digraph of components, written in multiple languages (C, Matlab, and FORTRAN), and enclosed in an optimization loop (see Fig. 1, left). Notice that information is exchanged between executions in three different languages and is streaming between the optimizer and the simulation. In addition, a variety of intermediate results are produced, not all of which are direct performance data. Such results are typically cached to improve performance, visualized at different stages of the execution, or simply saved for later inspection. Furthermore, the components (codes) are developed at different times by different researchers and many are still under

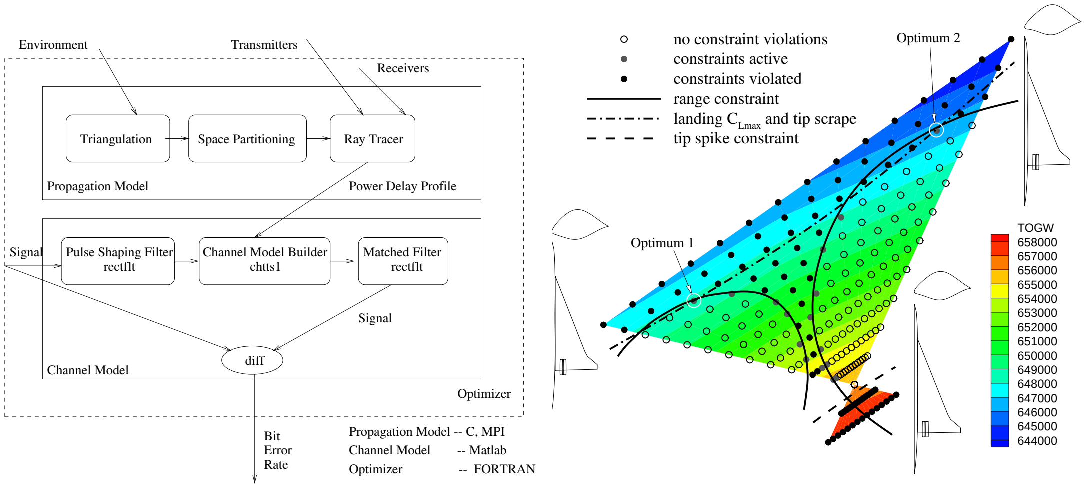


Figure 1: (left) Compositional modeling for designing a wireless communications system. (right) A slice of an aircraft configuration design space through three design points.

active development. Their I/O specifications hence cannot be enumerated in advance to achieve matching of components. Further, the possibilities of how components could be cascaded and combined can itself evolve over time. How can a programming environment be designed that allows the binding of problem specifications to arbitrary codes and allows their arbitrary composition?

- *Scenario 2:* A team of aircraft design engineers and numerical analysts are attempting to minimize the take-off gross weight (TOGW) for an aircraft configuration design involving 29 design variables with 68 constraints [42] (see Fig. 1, right). High-fidelity codes dealing with aerodynamics, mechanics, and geometry determine how changes in design variables affect the TOGW. This application domain is characterized not by an abundance of data, but rather by a scarcity of data (owing to the cost and time involved in conducting simulations). Consequently, the solution methodology involves a combination of high accuracy computations, surrogate modeling (to provide response surface approximations for unsampled design regions [56]) and a robust data management system to help focus data collection on the most promising regions. As a result, evaluating a design point might involve executing a high-fidelity computation, using low-fidelity approximations to obtain an estimate of the TOGW and/or querying a database to lookup previously conducted simulations. In addition, the resources for computing and data storage could be geographically distributed. How can a single environment provide unified access to such diverse facilities and what programming abstractions are available that allow its efficient and effective use?

- *Scenario 3:* A group of computer scientists, nuclear physicists, and performance engineers are modeling Sweep3D [57, 59], a complex ASCI benchmark for discrete ordinates neutron transport. They concur that efficient modeling of this application (over the 90% accuracy level) requires analytical modeling, simulation, and actual system execution paradigms simultaneously [2]! They use a metasystem infrastructure to combine these various models together in a unified manner. However, they are undecided over when to *switch* codes during the computation — do they use a low-level simulator for 80% of the available time, and then switch to analytic models or can they be confident of extrapolating using analytical models even earlier? What if excessive multi-threading on a machine leads to too many fluctuations in their estimates? What system architectures are available that enable compositional modeling when information about component choices is obtained during the computation (rather than before)?

1.2 Multidisciplinary Grid Communities: Themes

The key dominant theme in these scenarios, and one that is well accepted as an integral aspect of grid computing, is the ability to do **compositional modeling** [14, 54]. In the context of problem-solving, Forbus [30] defines this term as 'combining representations for different parts of a [computation] to create a representation of the [computation] as a whole.' In this paper, we employ this term to convey merely an approach to problem-solving and its use is not meant to imply an implementation technology, such as distributed object components (although that is one of the common ways of providing the functionality). For instance, a scientist explicitly moving input and output files across multiple program executables can be viewed as performing compositional modeling (albeit in a very primitive manner). Thus, a component could be any piece of software, executable, model fragment, or even a set of equations that helps the scientist to formalize the process of modeling a computation.

A second aspect (again, one whose assertion will hardly be controversial) is **collaboration**. By definition, a GCE for a grid community must support groups of scientists and engineers, not just single investigators. These users rely on each others' codes and data, contribute results to the total effort, communicate in a variety of ways, and organize themselves around subproblems in ways that are hard to predict. They may need to collaborate in real-time on a given simulation, but they are often at physically separate locations. Collaborative workspaces are fundamental to the way multidisciplinary research is conducted.

While the above two aspects are underscored in many grid projects, GCEs for multidisciplinary communities have a unique responsibility (and opportunity) to exploit the larger **context** of the scientific or engineering application, defined by the ongoing activities of the pertinent community. Typical GCEs only deal with one simulation at a time. The larger context we allude to here may include previous scientific results which can be used to improve the efficiency of current simulations or avoid computation altogether if a desired result is already available. The context may denote the fact that a simulation is being run as part of a higher-level problem-solving strategy, e.g., involving optimization or recommendation. Context also implies previous computational experience or performance, e.g., grid resources may be assigned more intelligently if the performance of previous similar simulations is known. A final example of context is the fact that a given simulation is often part of an *ensemble* of simulations; recognizing this aspect can help in creating more sophisticated simulation management tools.

As we will show below, the synergy resulting from consideration of all of the above three aspects (compositional modeling, collaboration, context) poses a unique set of research issues pertinent for multidisciplinary communities. An important goal of our approach is to maximize the synergy between grid computing on the one hand and multidisciplinary scientific problem-solving on the other. Thus, we are trying to answer two questions: (i) 'How can a multidisciplinary community setting be exploited to better use a grid?'; and (ii) 'How can the grid setting be exploited to better serve a scientific problem-solving community?'

1.3 GCEs for Multidisciplinary Grid Communities: Characteristics

Abstracting from the scenarios described above, and reflecting on the three themes just discussed, where do we locate multidisciplinary grid communities in the 'space' of computational grid users? To answer that question, we find that the following three dimensions are useful in characterizing GCEs for multidisciplinary communities. These dimensions should not be viewed as a one-to-one translation of the above themes into features; rather, they are the most pertinent forms of distinctions that will help us identify requirements for GCEs for multidisciplinary communities.

- **Emphasis on component coding effort versus component composition effort**

  Traditional programming environments emphasize either the coding of components (influenced by an implicit composition style) or the aspect of connecting them together (to prototype complex computations). For instance, when coding effort is paramount and composition is implemented in a distributed objects system (e.g., [22, 39]), techniques such as inheritance and templates can be used to create new components. Other implementations involving parallel programming [12, 19, 31] or multi-agent coordination [25, 26] provide comparable facilities (typically APIs) for creating new components. Component composition effort, on the other hand, emphasizes the modeling of a computation as a process of, say, graphically laying out a network of components (e.g., [52]). By providing a sufficiently rich vocabulary and database of primitive components, emphasis is shifted to composition rather than coding.

Design decisions made about component implementation and composition style indirectly influence the options available for composition and coding, respectively. This dimension distinguishes programming environments based on how they 'carve up' compositional modeling; which of these efforts do they emphasize more? By placing what forms of restrictions and assumptions on the other? In a multidisciplinary setting (e.g., *Scenario 1*), programming environments are required to emphasize both efforts in almost equal importance. The needs of the underlying application (in this example, wireless communications) render typical assumptions on both coding and composition style untenable.

- **Cognitive discordance among components**

  An indirect consequence of typical compositional modeling solutions is that they commit the scientist to an implementation (and representation) vocabulary. For example, components in LSA [39] (and most object-based implementations) are required to be high-performance C++ objects, instantiated from class definitions. This is not a serious constraint for typical grid communities since there is usually substantial agreement over the methodology of computation. The only sources of discordance here involve format conversions and adherence to standards (e.g., matrices in CSR format versus matrices in CSC format).

  In multidisciplinary grid communities (see *Scenarios 1 and 2*), there are huge differences in vocabulary (e.g., biologists, civil engineers, and economists using a watershed assessment PSE have almost no common terminology) and fundamental misunderstandings and disagreements about the way computations should be organized and modeled (e.g., aerodynamicists, control engineers, and structural engineers model an aircraft in qualitatively different ways). Furthermore composition in such a setting typically involves multiple legacy codes in native languages, and requires the ability to adjust to changing data formats, data sources (e.g., user-supplied, accessed through grid information services, streamed from another module etc.). Cognitive discordance is a serious issue here, one that is impossible to address by committing to a standard vocabulary for implementing components. Such messiness should be viewed not as a limiting bottleneck, but a fundamental aspect of how multidisciplinary research is conducted.

- **Sophistication of simulation management**

  Traditional GCEs make a simple-minded distinction between the representation of a component and its implementation, suitable for execution on the grid. Representation is usually intended to imply naming conventions and association of features (e.g, "is it gcc-2.0.8 compliant?") to help in execution. Once again, this has not proved a serious constraint since grid services have traditionally focused more on executing computations (single runs) and less on high-level problem solving. The sophistication of simulation management is directly related to the representational adequacy of components in the GCE.

  For situations such as described in *Scenarios 2 and 3*, the scientist would like to say "Conduct the same simulation as done on Friday, but update the response surface modeling to use the new numbers collected by Mark." Or perhaps, "collect data from parameterized sweeps of all performance models of the Sweep3D code where the MPI simulation fragment occupies no more than 30% of the total time." Simulation management can be viewed as a facility for both high-level specification of runs as well as a way to seamlessly *mix* computations and retrievals from a database of previously conducted simulations (see *Scenario 2*). This implies that data management facilities should be provided not as a separate layer of service, but as a fundamental mode by

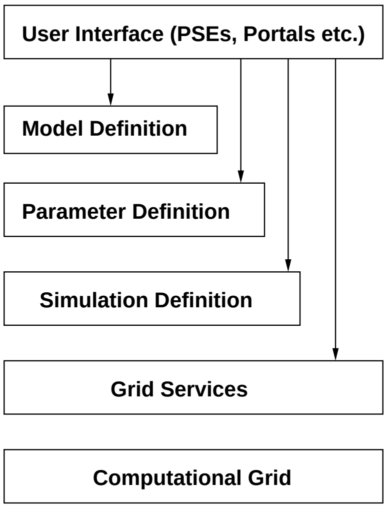


Figure 2: Layers of functionality needed to support multidisciplinary grid communities.

which the simulation environment in a GCE can be managed. The recent NSF-ITR-funded GriPhyN project [7] and the Sequoia data management system [74], both multidisciplinary endeavors, are motivated by similar goals. Simulation management also serves as a way of documenting 'history' of computational runs and experiments. For example, in conducting parameterized sweeps [17], knowing that certain particular choices have been executed elsewhere on the grid allows flexibility in load balancing and farming out computations to distributed resources.

1.4 GCEs for Multidisciplinary Grid Communities: A High-Level Architecture

Finally, by way of introduction, we present a high-level architecture or design framework for organizing and building GCEs for multidisciplinary grid communities (see Fig. 2). We believe that programming capabilities improve by recognizing modeling assumptions and explicitly factoring them out in a system design architecture. Fig. 2 does not describe an architecture in the full sense of the word, e.g., with precisely defined interfaces between layers. However, it does separate out the various functions or modes that must be represented in a powerful and effective multidisciplinary community GCE. The functional framework of the Grid summarized in Fig. 2 is complementary to ones that are based on protocol layering (see [35]) and commodity computing (see [37]).

Model: A model is a directed graph of specific executable pieces defining the control-flow and data-flow in a computation, e.g., the digraph in Fig. 1 (left). We distinguish between a model and its representation in a GCE; the representation might involve just the model's name or it might involve opening up the boxes (nodes in the digraph) and representing them in a more sophisticated fashion. Although models consist of ready-to-run pieces of code, these pieces may be parameterized.

Model Instance: A model instance is a model with all parameters specified. Note that some of these parameters may not be specified until runtime. Thus, while there might not exist a static conversion from models to model instances, the distinction between model instances and models is still useful. For example, using two different input data sets with the same model corresponds to two different model instances and a parameter sweep tool can be used to generate such model instances.

Simulation: A simulation is a model instance assigned to and run on a particular computational resource on the grid. It is useful to distinguish between a model instance and a simulation because, for example, a single model instance can be run (and re-run) many times using different computational resources or different random number sequences; each of these would be a new simulation by our conventions.

Given these definitions, the framework summarized in Fig. 2 can be used to organize the various functions which should be supported in a GCE for a typical multidisciplinary grid community. The *model definition* layer is where users who need to create or modify models find tools to support this activity. Users who simply use existing models require only trivial support from this layer. In the *parameter definition* layer we locate those activities that associate model instances with models. Examples include tools that generate parameter sweeps [17] or other types of model instance ensembles, as well as the use of problem-oriented scripting languages to generate multiple model instances. (Note that we are using 'parameter' in a very broad sense here, making no specific assumptions about exactly how these parameters are defined or what they include.) Another activity that is naturally found at the parameter definition level is a 'database query' mode, in which results from previous simulations are accessed, perhaps instead of doing new computations. The next layer, *simulation definition*, is where a model instance (or set of model instances) is assigned to grid resources. In the simplest case, a user simply chooses some subset of available grid resources to which the model instance should be mapped. More interesting, however, are the possibilities for simulation-management tools which take a set of model instances and assign them to the grid, perhaps with sophisticated load balancing strategies or leveraging performance summaries from previous simulations. The lowest two levels appearing in Fig. 2, *grid services* and *computational grid*, correspond to the software and hardware resources (e.g., Globus, networks, machines) that make computational grids possible. As mentioned earlier, our emphasis has been on high-level, application-specific issues. We omit further discussion of the architecture, protocols, and services being developed elsewhere for these levels (e.g., see [35]).

Note that not all services or activities fit neatly into the categories shown. For example, in computational steering [52], model parameters may be modified and computational resources re-assigned at runtime; so the parameter and simulation definition services are interleaved with execution in this setting. Other important aspects of an effective GCE are not explicitly represented in Fig. 2. For example, support for collaboration is implicit throughout. However, this high-level view of required layers of functionality helps organize and orthogonalize our efforts.

In keeping with the typical end-to-end design philosophy of the Grid [34], we have attempted to provide support for these new services as layers of abstraction over traditional low-level grid scheduling and resource management facilities. In addition, our resulting high-level architecture 'teases out' typically blurred layers into distinct levels at which various services can be provided. Three of our specific contributions to this architecture include (i) a lightweight data management system that supports compositional modeling (at the model definition level), helps view experiment evaluation as querying (at the parameter definition level), and provides bindings and semistructured representations (for all levels) (ii) a collaborative component composition workspace (*Sieve*) for model definition, and (iii) a framework for distributed resource control (*Symphony*) that provides core support for parameter and simulation definition and management. We describe these efforts in more detail in Section 3.

2 Motivating Applications

This section briefly describes five PSEs that are variously situated along the grid community characteristic axes (see Section 1.3). These examples highlight the diversity of multidisciplinary communities that a unifying GCE architecture must support.

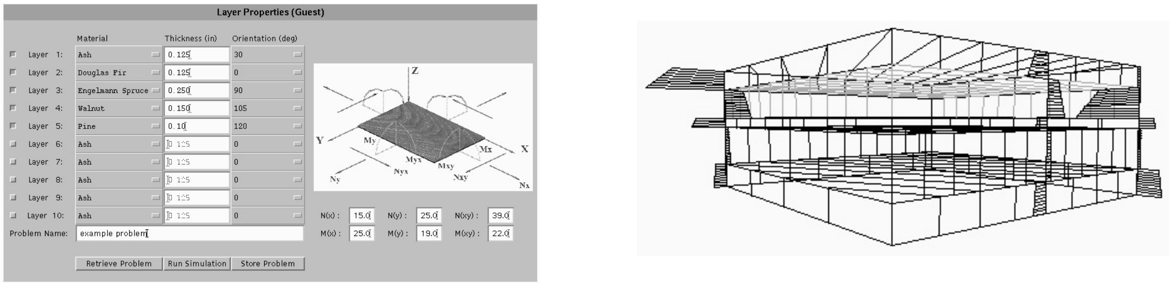


Figure 3: (left) Input interface to the CMA model in the WBCSim PSE [43]. (right) Wireframe model of a wood-based composite showing failed layers (gray) and active layers (black), and the orientation of fibers in each layer. In this figure, the second layer has failed.

2.1 WBCSim

WBCSim is a prototype PSE that is intended to increase the productivity of wood scientists conducting research on wood-based composite materials, by making legacy file-based FORTRAN programs, which solve scientific problems in the wood-based composites domain, widely accessible and easy to use. WBCSim currently provides Internet access to command-line driven simulations developed by the Wood-Based Composites (WBC) Program at Virginia Tech. WBCSim leverages the accessibility of the Web to make the simulations with legacy code available to scientists and engineers away from their laboratories. WBCSim integrates simulation codes with a graphical front end, an optimization tool, and a visualization tool. The system converts output from the simulations to the Virtual Reality Modeling Language (VRML) for visualizing simulation results. WBCSim has two design objectives: (1) to increase the productivity of the WBC research group by improving their software environment, and (2) to serve as a prototype for the design, construction, and evaluation of larger scale PSEs. The simulation codes used as test cases are written in FORTRAN 77 and have limited user interaction. All the data communication is done with specially formatted files, which makes the codes difficult to use. WBCSim hides all this behind a server and allows users to supply the input data graphically, execute the simulation remotely, and view the results in both textual and graphical formats.

WBCSim contains four simulation models of interest to scientists studying wood-based composite materials manufacturing — rotary dryer simulation (RDS), radio-frequency pressing (RFP), composite material analysis (CMA), and particle mat formation (MAT). The rotary dryer simulation model was developed as a tool to assist in the design of drying systems for wood particles, such as used in the manufacture of particleboard and strandboard products. The rotary dryer is used in about 90 percent of these processes. The radio-frequency pressing model was developed to simulate the consolidation of wood veneer into a laminated composite, where the energy needed for cure of the adhesive is supplied by a high-frequency electric field. The composite material analysis model was developed to assess the strength properties of laminated fiber reinforced materials, such as plywood. The mat formation model is used to calculate material properties of wood composites, modeling the mat formation process as wood flakes are deposited and then compressed into a mat. This model is crucial for all other manufacturing process models, as they require material properties as input.

The software architecture for WBCSim is three-tiered: (i) the legacy simulations and various visualization and optimization tools, perhaps running on remote computers; (ii) the user interface; and (iii) the middleware that coordinates requests from the user to the legacy simulations and tools, and the resulting output. These three tiers are referred to as the developer layer, the client layer, and the server layer, respectively. The developer layer consists primarily of the legacy codes on which WBCSim is based. The server layer expects a program in the developer layer to communicate its data (input and output) in a certain format. Thus, legacy programs are ‘wrapped’ with custom Perl scripts, and each legacy program must have its own wrapper. The client layer consists of Java applets and is responsible for the user interface (see Fig. 3, left). It also handles communication with the server layer, is the

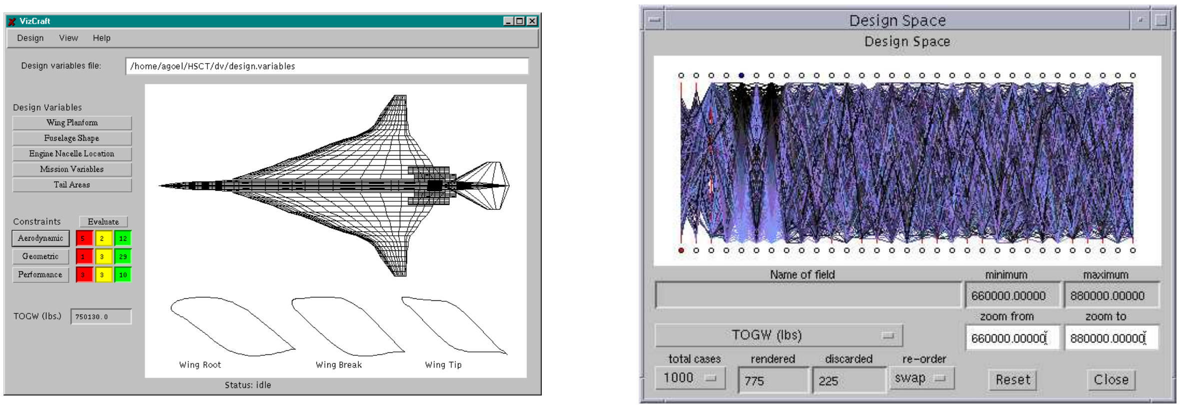


Figure 4: (left) VizCraft design view window showing aircraft geometry and cross sections [42]. (right) Visualizing 156 aircraft design points in 29 dimensions with a careful assignment of variables to color drivers reveals an interesting association.

only layer that is visible to end-users, and typically will be the only layer running on the user's local machine. The server layer is the core of WBCSim as a system distinct from its legacy code simulations and associated data viewers. The server layer is responsible for managing execution of the simulations and for communicating with the user interface contained in the client layer. WBCSim applications require sophisticated management of the execution environment; the server layer, written in Java, directs execution of multiple simulations, accepts multiple requests from clients concurrently, and captures and processes messages that signify major milestones in the execution (such as the computation of an intermediate value). Graphical results from the simulations are communicated to the clients using an HTTP server (see Fig. 3, right).

2.2 VizCraft

VizCraft [42] is a PSE that aids aircraft designers during the conceptual design stage. At this stage, an aircraft design is defined by a vector of 10 to 30 parameters. The goal is to find a vector that minimizes a performance-based objective function while meeting a series of constraints. VizCraft integrates simulation codes to evaluate a design with visualizations for analyzing a design individually or in contrast to other designs. VizCraft allows the designer to easily switch between the view of a design in the form of a parameter set, and a visualization of the corresponding aircraft geometry. The user can easily see which, if any, constraints are violated. VizCraft also allows the user to view a database of designs using the parallel coordinates visualization technique. VizCraft is a design tool for the conceptual phase of aircraft design whose goal is to provide an environment in which visualization and computation are combined. The designer is encouraged to think in terms of the overall task of solving a problem, not simply using the visualization to view the results of the computation.

VizCraft provides a menu-driven graphical user interface to the high speed civil transport (HSCT) design code that uses 29 variables and 68 realistic constraints. This code is a large (million line) collection of C and FORTRAN routines that calculate the aircraft geometry in 3-D, the design constraint values, and the take-off gross weight (TOGW) value, among other things. VizCraft displays the HSCT planform (a top view), cross sections of the airfoil at the root, leading edge break, and tip of the wing, and color coded (red, yellow, green) constraint violation information. To help manage the large number of constraints, they are grouped conceptually as aerodynamic, geometric, and performance constraints. Design points, and their corresponding TOGW, are displayed via active parallel coordinates. The parallel coordinates are also color coded, and they can be individually scaled, reordered, brushed, zoomed, and colored. A parallel coordinate display for the constraints can be similarly manipulated. While the inte-

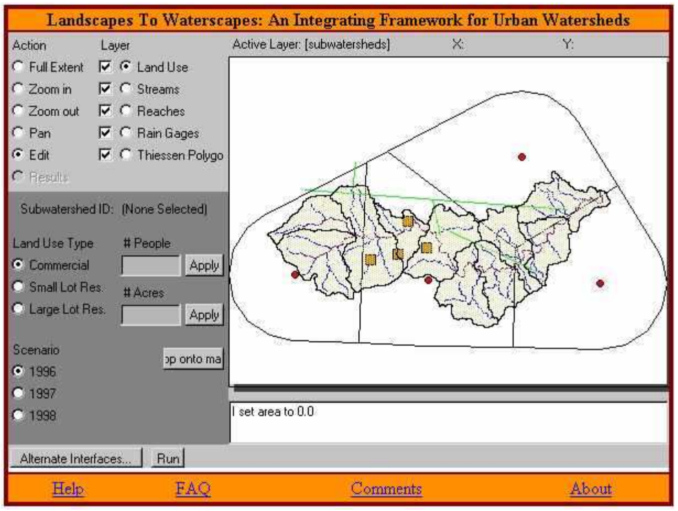


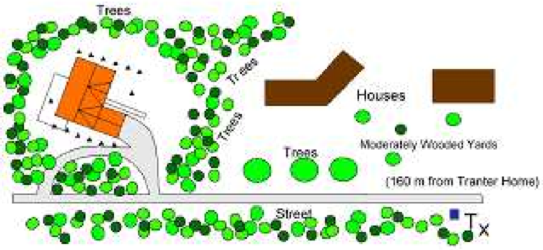
Figure 5: Front-end decision maker interface to the L2W PSE [70], depicting landuse segmentation of the Upper Roanoke River Watershed in Southwest Virginia.

gration of the legacy multidisciplinary HSCT code into a PSE is nontrivial, the strength and uniqueness of VizCraft lie in its support for visualization of high dimensional data (see Fig. 4).

2.3 L2W

Landscapes to Waterscapes (L2W) is a PSE for landuse change analysis and watershed management. L2W organizes and unifies the diverse collection of software typically associated with ecosystem models (hydrological, economic, and biological), providing a web-based interface for potential watershed managers and other users to explore meaningful alternative land development and management scenarios and view their hydrological, ecological, and economic impacts. Watershed management is a broad concept entailing the plans, policies, and activities used to control water and related resources and processes in a given watershed. The fundamental drivers of change are modifications to landuse and settlement patterns, which affect surface and ground waterflows, water quality, wildlife habitat, economic value of the land and infrastructure (directly due to the change itself such as building a housing development, and indirectly due to the effects of the change, such as increased flooding), and cause economic effects on municipalities (taxes raised versus services provided). The ambitious goal of L2W is to model the effects of landuse and settlement changes by, at a minimum, integrating codes/procedures related to surface and subsurface hydrology, economics, and biology. The development of L2W raises issues far beyond the technical software details, since the cognitive discordance between computer scientists (developing the PSE), civil engineers (surface and subsurface hydrology), economists (land value, taxes, public services), and biologists (water quality, wildlife habitat, species survival) is enormous. The disparity between scientific paradigms in a multidisciplinary engineering design project involving, say, fluid dynamicists, structural, and control engineers is not nearly as significant as that between computer scientists, civil engineers, economists, and biologists. A further compounding factor is that L2W should also be usable by governmental planners and public officials, yet another different set of users.

The architecture of the L2W PSE is based on leveraging existing software tools for hydrology, economic, and biological models into one integrated system. Geographic information system (GIS) data and techniques merge both the hydrologic and economic models with an intuitive web-based user interface. Incorporation of the GIS techniques into the PSE produces a more realistic, site-specific application where a user can create a landuse change scenario based on local spatial characteristics (see Fig. 5). Another advantage of using a GIS with the PSE is that the GIS can



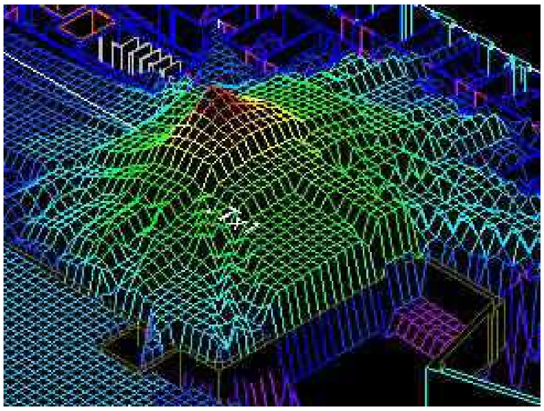


Figure 6: (left) Example outdoor environment for designing a wireless communications system in the S^{4}W PSE [51]. (right) Propagation coverage prediction around the region of interest in the environment.

obtain necessary parameters for hydrologic and other modeling processes through analysis of terrain, land cover, and other features. Of all the PSEs described here, L2W is unique in that it is centered around a GIS. Currently, L2W integrates surface hydrology codes and economic models for assessing the effect of introducing settlement patterns. Wildlife and fisheries biologists were involved in the L2W project, but their data and models are not fully integrated as of this writing. The biological models include the effect of development on riparian vegetation, water quality, and fish and wildlife species.

2.4 S^{4}W

S^{4}W ('Site-Specific System Simulator for Wireless Communications') is a collaborative PSE for the design and analysis of wideband wireless communications systems. In contrast to the above described projects, the S^{4}W project is occurring in parallel with the development of high-fidelity propagation and channel models; this poses a unique set of requirements for software system design and implementation (ref. *Scenario 1* in the introduction) [78]. S^{4}W has the ability to import a 3-dimensional database representing a specific site (see Fig. 6, left), and permits a wide range of radio propagation models to be used for practical communications scenarios [51]. For example, in a commercial wireless deployment, there is a need to budget resources, such as radio channel assignments and the number of transmitters. S^{4}W allows wireless engineers to automatically drive the simulation models to maximize coverage or capacity, or to minimize cost. Furthermore, unlike existing tools, S^{4}W permits the user to import measured radio data from the field, and to use this data to improve the models used in the simulation. A knowledge-based recommender system [65] provides improved modeling capability as the software corrects the environment model and the parameters in the propagation model, based on measured data. Finally, the ability to optimize the location of particular wireless portals in an arbitrary environment is a fundamental breakthrough for wireless deployment, and S^{4}W has the ability to perform optimization based on a criterion of coverage, QoS, or cost (see Fig. 6, right).

While primitive software tools exist for cellular and PCS system design, none of these tools include models adequate to simulate broadband wireless systems, nor do they model the multipath effects due to buildings and other man-made objects. Furthermore, currently available tools do not adequately allow the inclusion of new models into the system, visualization of results produced by the models, integration of optimization loops around the models, validation of models by comparison with field measurements, and management of the results produced by a large series of experiments. One of the major contributions of S^{4}W is a lightweight data management subsystem [78] that supports the experiment definition, data acquisition, data analysis, and inference processes in wireless system design. In particular, this facility helps manage the execution environment, binds representations to appropriate

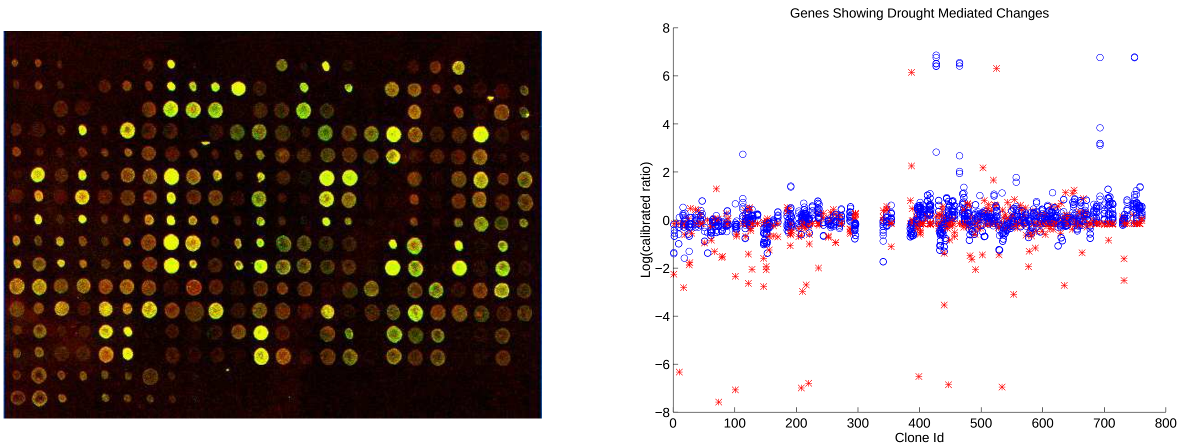


Figure 7: An example microarray design in Expresso [6] to study gene expression in Loblolly pine clones. (left) The microarray is printed in four 24 × 16 sub-quadrants, one of which is shown here. Figure courtesy Y.-H. Sun (NCSU). (right) Expresso output depicting 265 clones (out of a total of 768) that responded to three cycles of mild drought stress.

implementations in a scientific computing language, and aids in reasoning about models and model instances.

Supported by a $1M grant from the NSF Next Generation Software program, S^4W is designed to enhance three different kinds of performance — software, product, and designer. Superior software performance is addressed in this project by (i) developing fundamentally better wireless communication models, (ii) constructing better simulation systems composed from the component wireless models via the recommender, and (iii) the transparent use of parallel high-performance computing hardware via the composition environment's access to distributed resources. Superior product performance (the actual deployed wireless systems) is addressed by using optimization to design *optimal* rather than merely *feasible* systems. Superior designer performance is directly addressed by the synergy resulting from the integrated PSE, whose purpose is to improve designer performance and productivity.

2.5 Expresso

The Expresso project [6] addresses the entire lifecycle of microarray bioinformatics, an area where 'computing tools coupled with sophisticated engineering devices [can] facilitate discovery in specialized areas [such as genetics, environment, and drug design]' [45]. Microarrays (sometimes referred to as *DNA chips*) are a relatively new technique in bioinformatics, inspired by miniaturization trends in micro-electronics. Microarray technology is an experimental approach to study all the genes in a given organism simultaneously; it has rapidly emerged as a major tool of investigation in experimental biology. The basic idea is to 'print' DNA templates (*targets*), for all available genes that can be expressed in a given organism, onto a high-density 2D array in a very small area on a solid surface. The goal then is to determine the genes that are expressed when cells are exposed to experimental conditions, such as drought, stress, or toxic chemicals. To accomplish this, RNA molecules (*probes*) are extracted from the exposed cells and 'transcribed' to form complementary DNA (cDNA) molecules. These molecules are then allowed to bind (*hybridize*) with the targets on the microarray and will adhere only with the locations on the array corresponding to their DNA templates. Typically such cDNA molecules are tagged with fluorescent dyes, so the expression pattern can be readily visualized as an image. Intensity differences in spots will then correspond to differences in expression levels for particular genes. Using this approach, one can 'measure transcripts from thousands of genes in a single afternoon' [45]. Microarrays thus constitute an approach of great economic and scientific importance, one whose methodologies are continually evolving to achieve higher value and to fit new uses.

The Expresso PSE [6] is designed to support all microarray activities including experiment design, data acquisition, image processing, statistical analysis, and data mining. Expresso's design incorporates models of biophysical and biochemical processes (to drive experiment management). Sophisticated codes from robotics, physical chemistry, and molecular biology are 'pushed' deeper into the computational pipeline. Once designs for experiments are configured, Expresso continually adapts the various stages of a microarray experiment, monitoring their progress, and using runtime information to make recommendations about the continued execution of various stages. Currently, prototypes of the latter three stages of image processing, statistical analysis, and data mining are completely automated and integrated within our implementation.

Expresso's design underscores the importance of modeling both physical and computational flows through a pipeline to aid in biological model refinement and hypothesis generation. It provides for a constantly changing scenario (in terms of data, schema, and the nature of experiments conducted). The ability to provide expressive and high performance access to objects and streams (for experiment management) with minimal overhead (in terms of traditional database functionality such as transaction processing and integrity maintenance) [44] is thus paramount in Expresso.

The design, analysis, and data mining activities in microarray analysis are strongly interactive and iterative. Expresso thus utilizes a lightweight data model to intelligently 'close the loop' and address both experiment design and data analysis. The system organizes a database of problem instances and simulations dynamically, and uses data mining to better *focus* future experimental runs based on results from similar situations. Expresso also uses inductive logic programming (ILP), a relational data mining technique, to model interactions among genes and to evaluate and refine hypothesized gene regulatory networks. One complete instance of the many stages in Expresso has been utilized to study gene expression patterns in Loblolly pine [46], in a joint project with the Forest Biotechnology group of North Carolina State University.

3 Systems Support for Multidisciplinary GCE Applications

This section describes several core systems support technologies useful for developing GCEs (and currently employed in the applications outlined so far). Many of these tools and frameworks rely on the notion of representations of components; we begin by motivating this idea.

3.1 Representations in a GCE

One of the main research issues in GCEs is modeling the fundamental processes by which knowledge about scientific models is created, validated, and communicated. As mentioned in Section 1 and illustrated in the many example systems of Section 2, the expressiveness with which a scientist could interact with a GCE is directly related to the adequacy of representation provided by the system. While it is true that there is no universal representation that is ideal for all purposes, traditional approaches employed in grid projects (for representing models, for model instances, and for simulations) are very restrictive.

Recall that we defined a model to denote a directed graph of specific computational codes or executables. The notion of the 'representation of a model' is open to many interpretations and intensely debated in the modeling literature (see for instance [29]); we will not attempt to settle this debate here. Instead, we adopt an operational definition for the representation of a model, namely that it is an abstraction of the model that permits useful problem-solving capabilities that would not be possible with the model alone. The abstraction could refer to the functional behavior of the model (e.g., a signature), the structural constituents of the model (e.g., a digraph of model fragments), a profile of its performance (to aid in design and analysis), its relationships to other models, and/or information about how it fits within the larger computational context of the GCE and the activities conducted within it.

Consider two extremes of representing a single computational component (the simplest model) in a GCE. A black-box representation would be one where just the name of the component serves as its representation. Such a solution will definitely aid in referring to the component over the grid (e.g., 'run the XYZ software'), but doesn't help in any further sophisticated reasoning about the component (.g., 'is XYZ an iterative algorithm?') At the other extreme, a white-box representation is one where the component itself serves as its representation (for example, mathematical descriptions of scientific phenomena). Usually in such cases, the representation is accompanied by a domain theory and algorithms for reasoning in that representation. For instance, the QSIM system [58] is a facility where once a component (e.g., one for the cooling of liquid in a cup [30]) is defined, it is possible to reason about the properties and performance of that component (e.g., when are the contents of the cup drinkable?). While extremely powerful, such designs work well only within restrictive (and sometimes artificial) applications. An intermediate solution is to annotate components with (feature, value) pairs describing attribute-based properties. For instance, annotations might involve directives, flags, and hints for compiling the code on a specific platform.

These issues are amplified when we consider the model to be a digraph of computational components. While many projects distinguish between models and representations, two main approaches can be distinguished here. In the first category, representations are motivated by the need to manage the execution environment (e.g., 'schedule this graph on the grid, taking care to ensure that data files are properly matched'). Examples here are the Linear System Analyzer (LSA) [40] and the Component Architecture Toolkit (CAT) [13] at Indiana University, the ZOO desktop experiment management environment at the University of Wisconsin [50], the Application Visualization System of Advanced Visual Systems, Inc. [76], and the SCIRun computational steering system at the University of Utah [52]. Projects in the second category are based on AI research in compositional modeling [28, 62, 69] and are motivated by formal methods for reasoning with (approximate and qualitative) representations. The modeling and performance requirements in a multidisciplinary GCE mean that both approaches are too restrictive.

With the advent of XML and the emergence of the Web as a large-scale semistructured data source, interest in semistructured representations has expanded into the GCE/PSE community. A plethora of markup languages, XML-based formats, and OO coding templates have been proposed for representing aspects of domain-specific scientific codes (e.g., SIDL [21]). In addition, a variety of formats have been proposed recently (e.g., SOX [38]) for defining metadata associated with various stages of computational experiments [38]. A major advantage of such solutions is that the ensuing representations can be cataloged and managed by database technology.

Our goal is to investigate representations that (i) allow the binding of problem specifications to models (and model instances), without insisting on an implementation vocabulary (for the models); (ii) can help us to reason both about the models being executed as well as data produced from such simulations; and (iii) help design facilities such as change management, high-level specification of simulations, recommendation, optimization, and reasoning (about models and model instances).

3.2 BSML: A Binding Schema Markup Language

Akin to other GCE projects, our emphasis here will be on semistructured representations for models. However we view markup languages such as XML as less of a data format, programming convention, or even a high-level abstraction of a programming environment. Rather, we view them as a vehicle to define *bindings* from representations to models in a GCE. Binding refers to the process of converting XML data to an appropriate encoding in a scientific computing language (the reverse process is fairly straightforward). There are several forms of bindings in a GCE — binding of values to language variables, converting an XML format to some native format that can be read directly by the model, and/or generating source code for a stub that contains embedded data and a call to the appropriate language function using these data as parameters. Notice that we do not make a distinction between invoking a component procedurally in a scientific computing language, generating code that invokes a component, or executing a program with command line arguments. All of these are bindings from one representation to various assumptions on the execution environment (which is presumably being handled by the existing computational setup). Our lack

```
<element name='pdp'>
 <sequence>
  <element name='rmsDelaySpread'
                       type='double'/>
  <element name='meanExcessDelay'
                       type='double'/>
  <element name='peakPower'
                       type='double'/>
  <code component="optimizer">
   <bind>print "$peakPower\n"</bind>
  </code>
 </sequence>
 <repetition>
  <sequence>
   <element name='time' type='double'/>
   <element name='power' type='double'/>
   <code component="chtts1|chttm">
    <bind>print " $time $power\n"</bind>
   </code>
  </sequence>
 </repetition>
 <code component="chtts1|chttm">
  <begin>print "M = [\n"</begin>
  <end>print "];\n"</end>
 </code>
</element>
```

Figure 8: BSML descriptions for a class of XML documents pertaining to power delay profiles (PDPs) in the S^4W PSE. Sample bindings for MATLAB are shown by the `bind` tags.

of any stringent assumptions on the computational codes or method of invoking models is fundamental to multidisciplinary research. From the viewpoint of the GCE, a single representation could be stored but which can allow all these forms of bindings to be performed.

A full description of our BSML (Binding Schema Markup Language) is beyond the scope of this article (for more details, see [78]). We briefly mention that BSML associates user-specified blocks of code with user-specified blocks of an XML file. ‘Blocks’ can be primitive datatypes, sequences, selections, and/or repetitions. Intuitively, primitive datatypes denote single values, such as double precision numbers; sequences denote structures; selections denote multiple choices of conveying the same information; and repetitions denote lists. While not particularly expressive, this notation is meaningful to GCE component developers, simple and efficient to implement, and general enough to allow the building of more complex data representations.

Consider, for example, representing a power delay profile (PDP) from the S^4W project in XML. A PDP is a two-column table that describes the power received at a particular location during a specified time interval. Statistical aggregates derived from power delay profiles are used, for example, to optimize transmitter placement in S^4W. We can use BSML to define bindings between PDPs and all applicable models in S^4W. Applying a parser generated from such a BSML document (see Fig. 8 for an example) to a PDP will yield native code in the underlying execution environment (in this case, an executable Matlab script that contains embedded data). For a different native execution environment, a different binding could be defined (for the same data type). Hence, our representation is truly semistructured.

Notice that we can rapidly prototype new model instances with this technique. Similarly, we can use the same BSML source to provide bindings for an optimizer (for locating base stations, see *Scenario 1* in Section 1). The feedback will be a sequence of peak powers, one number per line. Some twenty five lines of BSML source can therefore take care of data interchange problems for three components. Storing these PDPs in a database is also

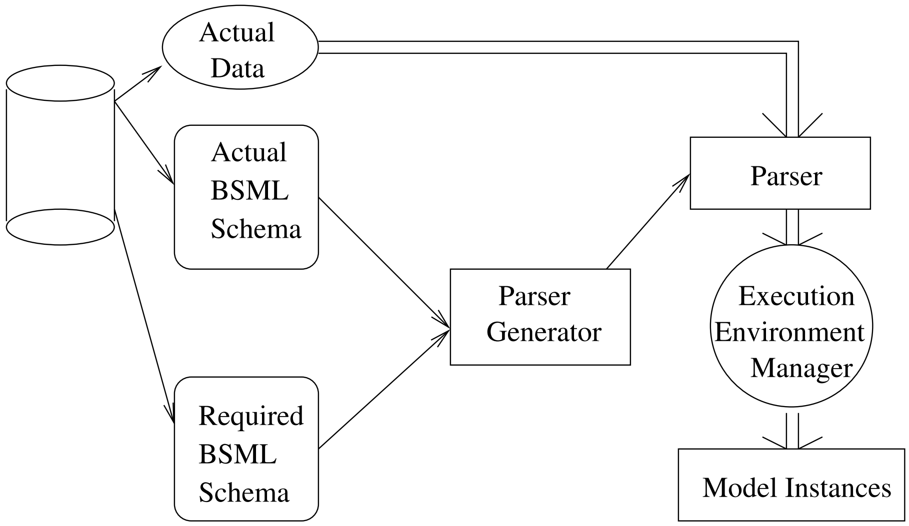


Figure 9: A facility for creating model instances from specifications represented in a Binding Schema Markup Language (BSML).

facilitated. From a system point of view, the schemas are the metadata and the software that translates and converts schemas is the parser generator. Figure 9 shows a typical configuration.

Both the data and the metadata are stored in a database. A parser is lazily generated for each combination of model's input port and the actual BSML schema whose data instance is connected to that port. Model descriptions can also be stored in the database. They consist of model id, description, schemas of its input and output ports, various execution parameters, and annotations, such as relations to other models (see Section 3.5). We do not provide any tools for checking the consistency of the output data with the output schemas because, unlike in Web or business domains, this is rarely an issue. A GCE model's output schema is rigid and does not depend on the actual input schema.

The execution environment manager (see Fig. 9) glues the generated parsers to the components. For full-featured languages like FORTRAN, it will simply link the parser with the model's code. Prototyping for languages like Matlab requires more attention. The output of the parser for such languages is typically the source code that needs to be supplied to the interpreter. The exact way the parsers are linked to the model is specified by the model's execution parameters. From this point, each model together with a set of generated parsers looks like a program that takes a number of XML streams as inputs and produces a number of XML streams as outputs. This is an appropriate representation for the management of the execution environment. Our goal is similar to those in [4, 27, 41] in that the common motivation is management of the execution environment; at the same time, our concern with high-level problem-solving (see next three sections) shifts the emphasis from a unifying programming environment to one that allows a data-driven approach to managing large-scale multidisciplinary codes.

Finally, it is relatively straightforward to store any resulting data from such processes in a database system. If an RDBMS is used, we can reuse BSML to generate a number of SQL update statements in the same manner we used it to generate a Matlab script in Fig. 8. One of these 'models' will then connect to the database and execute these statements. This is no different from other format conversions happening in the GCE.

3.3 Format Conversions and Change Management

One of the benefits of semistructured data representations is automatic format conversion. This feature is useful in the following situations: (i) A model is changed over time, but data corresponding to the older versions has already been recorded in the database system. An example from S^4W is the evolution of the space partitioning parameters in the ray tracer. After we have realized that placing polygons at the internal nodes of the octree can improve space usage by an order of magnitude, more parameters have been added to space partitioning; (ii) Several components need essentially the same parameters, but are not truly plug-and-play interchangeable. Minor massaging is necessary in order to make their I/O specifications match.

We model the following changes: insertions, deletions, replacements, and unit conversions. Insertions and deletions correspond to additions and removals of parameters. For example, a moving channel builder takes the same inputs as a static one, plus the velocity of the receiver. Thus, any input to a moving channel builder can be converted to the input to a static one by projecting out the receiver's velocity. Replacements represent changes in parameter representation, such as a conversion between spherical and rectangular coordinates. Unit conversions are a special case of conversions that are quite common and can be easily automated, such as conversions between watts and decibel milliwatts. Unit conversion can be performed by equation and constraint solvers [30].

In our XML representation, insertions can be handled by requiring default values for new parameters. Removals amount to deleting the old values. Replacements and unit conversions require user-supplied or automatically generated conversion filters. The modeling literature abounds in such conversions, but it is important to realize that conversion facilities are ad-hoc by nature, and therefore only work for small changes in the schema. Typically, it is not necessary to find a globally optimal conversion sequence. A thorough treatment of change detection can be found in [20].

Change management can also be used to realize any problem-solving feature that involves transforming between semistructured representations. For example, consider the possibility that two students configure a GCE independently with different choices for various stages in a computational pipeline and arrive at contradictory results. They could then query the database for 'What is different between the experiments that produced data in directory A from the ones in directory B'? — providing responses such as 'The only difference is that a calibration threshold of 0.84 was used in B instead of 0.96 for A,' which are obtained by automatically analyzing the XML descriptions [1]. Change detection and processing is crucial in several GCE projects, such as Expresso (see Section 2.5) where objects of interest change formats, stations, and schema rapidly.

3.4 Executing Simulations = Querying

Recall that we defined a simulation as a model instantiated with inputs, along with an assigned computational resource. This captures the notion of applying multiple models to multiple inputs to generate a database of simulation results and performance data. In this section, we describe how our semistructured representations of models and bindings can aid in even higher level problem-solving facilities. In particular, we concentrate on facilities such as the 'parameter sweep' tool [17] and the 'database query' mode (found in the parameter definition layer of Fig. 2). The facilities described in this section (i) produce model instances and also (ii) associate data generated from simulations back with the corresponding model instances. We gloss over the aspect of how simulations corresponding to a model instance are actually executed, since they are addressed in Section 3.7. In particular, when we refer to executing a simulation we imply that some assignment of computational resources to model instances has been done by the simulation definition layer (see Fig. 2).

In the database paradigm, a model instance can be represented as a view. Executing the simulation corresponds to materializing the view. The query behind the view is a join over models and data. To be meaningful, a simulation must further satisfy some *syntactic* and/or *semantic* constraints. Syntactic constraints ensure that the simulation can indeed be executed. Each simulation run must be given enough data and the data must conform to the appropriate

```
<experiment id='diff. prop.'>
 WHERE <environment id='$id'>
  <meta><type>urban</type></meta>
 </environment> CONTENT_AS $env IN "envs"
 CONSTRUCT <experiment id='diff. prop.: $id'>
  <model>...</model>
  <inputs>
   <input>$env</input>
   ...
  </inputs>
  <outputs>...</outputs>
 </experiment>
</experiment>
```

Figure 10: Constructing new XML data (to recalculate PDPs in S^4W) using the XML-QL query notation. WHERE..CONSTRUCT is the format for expressing queries in this language. Notice that the query is parameterized by the `$env` variable whose `type` is restricted to be `urban`.

schemas. Semantic constraints ensure that the models are meaningful in the specific problem domain. We will describe semantic constraints in the next section. In our framework, users can impose custom constraints, such as 'use only the datasets from last week.' Specifying a model instance therefore maps naturally into a database query. This feature also supports the provision of iteration, aggregation, and composition operators by introducing minimal overhead in implementation. For example, compositions can be achieved by relational joins, aggregation by user-defined VIEWs, and iteration by 'index striding' on domain-specific records.

Consider the following scenario. An S^4W developer of ray tracing propagation models has added a model that takes diffraction into account. She now wants to recalculate the PDPs for the environments where diffraction is most significant, e.g., for urban outdoor environments. Her request for new simulations can be specified in a query notation such as XML-QL (see Fig. 10 for how this can be done). The result of this query is a sequence of model instances, which can be associated with corresponding simulations and scheduled for execution. Not only is this form of specification concise, it also enables us to use well-known query optimization techniques to push costly operations 'deeper' into the computational pipeline [47]. In particular, query-based representations of model instances lose the distinction between **conducting a simulation to collect data** and **looking up (already simulated) data from a database**. Coupled with grid information services, such a representation can help determine if specified simulations have been conducted elsewhere on the grid, to avoid duplication of effort. Furthermore, 'nearby' simulations can be retrieved to construct a surrogate function for the entire simulation or parts of it, thereby replacing costly executions with cheap surrogate function evaluations. In large scale, multidisciplinary, engineering design the construction and use of such surrogates has become standard practice. The query approach facilitates the automatic construction and validation of surrogates or functional approximations.

In our current implementation the above two modes — simulation by executing a code and simulation by querying — are provided as distinct services. Before we can seamlessly mix computations with retrievals from a database, a logic for such a facility has to be defined. For example, do we adopt a policy that looks for cached results before spawning out simulations? Or is it a best-search effort for doing costly operations on external clusters and conducting smaller simulations locally? Or does it use different fidelity approximations (see Section 3.5) for different ranges of the parameter sweep? The design of these policies (along with their associated business and organizational ramifications) will influence the acceptability of the 'execution as querying' viewpoint.

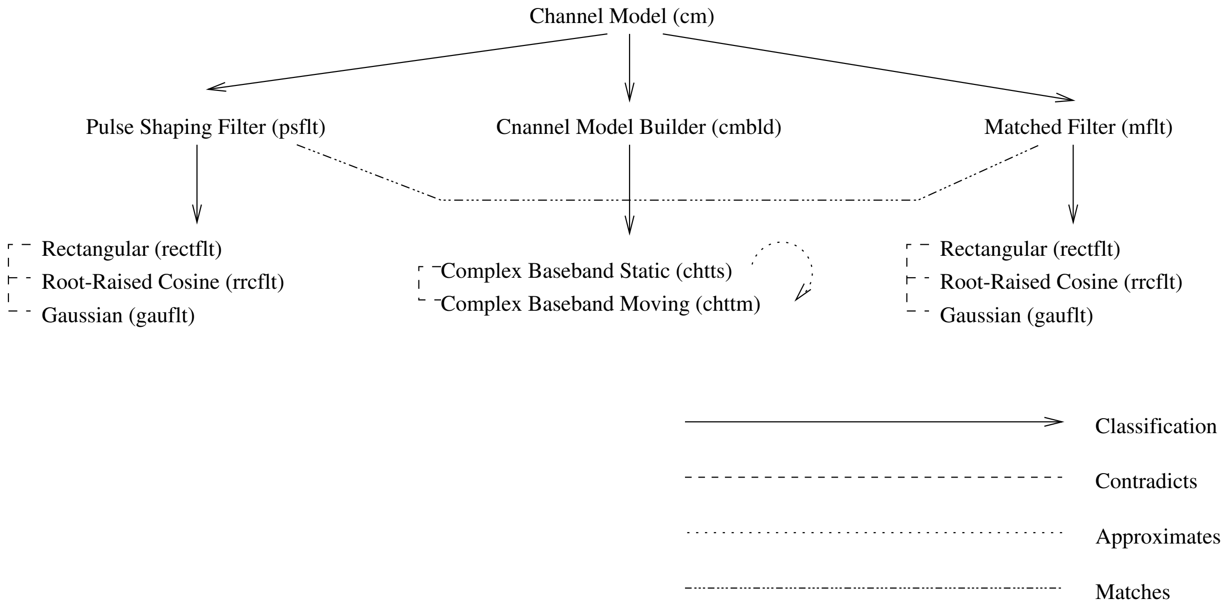


Figure 11: Relations between channel modeling components in S^4W.

3.5 Reasoning and Problem Solving

What constitutes a good model? GCEs should provide the facility to reason about a model and its constituent parts in terms of the features of the problem being solved and the desired performance constraints. For instance, this might help in recommending models for new problem instances, based on representational goals and performance criteria. A lot of domain-specific knowledge [62] is required to arrive at promising model choices, but a few general rules can be outlined.

First, a model must not contain any components that make incompatible assumptions about the phenomena being modeled. Following Nayak [62], we call such components *contradictory*. An example of contradictory components in S^4W is a class of model builders (static and moving). Second, some modeling choices may constrain the form of the rest of the model. For example, the signal filters of the transmitter and the receiver must *match*. And finally, models in a given class, say filters, often support similar forms of reasoning. We use the term *classification* to describe this aspect.

An example of these relations in S^4W is given in Fig. 11. The labels represent a small model library and the links represent the relations. Note that these relations are domain-specific and cannot be derived from the source in any general-purpose language. They must be supplied by the user (in this case, wireless system designer) as annotations to components. Such relations can then be used to prune the search space for recommendation and problem-solving.

An alternate, data-driven approach to reasoning is described in [48, 66, 67], where we have explored the knowledge-based selection of solution components for individual application domains. Such 'recommender systems' [63] can help in the natural process of a scientist/engineer making selections among various choices of algorithms/software in a GCE. They are typically designed off-line by organizing a battery of benchmark problems and algorithm executions [48], and mining it to obtain high-level rules that can form the basis of a recommendation. For instance, Fig. 12 describes a situation where data mining reveals regions of preference for one PDE algorithm over another, by mining a database of PDE solves. Visualization of performance data from scientific applications is also related to this aspect [72]. The importance of data mining in providing decision support in large-scale simulations [18] has also been recognized in other projects [32, 53, 60, 61, 64, 68]. Our framework is novel in that it

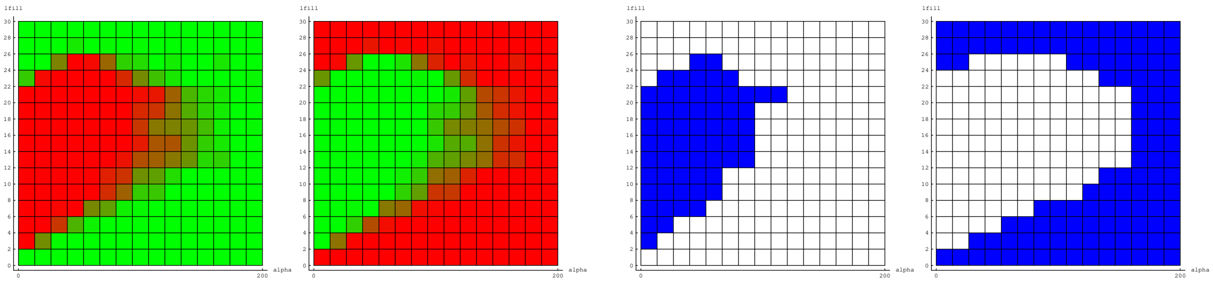


Figure 12: Mining and visualizing recommendation spaces for selecting among `gmres` (first picture) and direct solver (second) on nearly singular PDE problems; The x- and y-axes of the pictures denote problem and method parameters along which performances of the solvers are characterized. The intensity in the colormap reflects the number of problem instances in a bucket for which the given algorithm was superior; 90% confidence regions mined automatically for `gmres` (third) and direct solve (fourth), by the method described in [66] are also shown. For these regions, the given methods were superior for at least 90% of the problem instances.

captures the entire problem-solving process prior to the scheduling of a simulation and the semistructured format for model instances allows the embedding of mining functions as primitives into standard query languages (akin to [75]). It also facilitates the incorporation of performance models of the various parts of a computation [3] in reasoning and recommendation.

3.6 Sieve: A Collaborative Component Composition Workspace

Readers familiar with components and distributed internet-based applications will recognize that many goals of GCEs described earlier apply to other distributed applications. While the details differ, describing a computation as a network of components and providing access to a database (in this case the database of experimental runs) are not unique to computational science. While supporting legacy code is often central to computational science applications, this need is also not novel.

However, the combination of issues embodied in multidisciplinary GCE settings presents novel problems. These include the fact that individual runs of a simulation can take hours; the extensive and integrated use of visualization; the inherently distributed nature of the computation (i.e., certain sub-models may need to run on differing systems for reasons related to resource needs, or simply because they are legacy codes written for differing systems); the desire for synchronous collaboration; and the needs of multidisciplinary users, no one of whom is an expert in all aspects of the larger system (recall the L2W PSE from Section 2.3).

We embody the model definition activities of a GCE in a (collaborative) visual workspace, in which the user places various objects. These objects are components that represent individual codes, optimization tools, visualization software, etc. These components are linked together by the user to form networks that indicate the flow of data or control. The links between components are often represented by arrows. For example, a component representing an input file on some computer might be linked by an arrow to another component representing a model/optimizer combination. Another arrow links the model/optimizer combination to a visualizer. The intent is that the GCE will cause the input file to be moved to the machine storing the model and optimizer, and the model/optimizer will then be invoked. The output of this process will then be passed to the visualization, (perhaps on another machine) with the results displayed on the user's screen. The fundamental interface design is similar to that of a modular visualization environment (MVE) or the Khoros image processing system [80].

Our primary tool for creating interfaces for model building is named Sieve. Sieve provides a collaborative workspace that supports the deployment of of JavaBeans. At its heart, Sieve is a collaborative "Java Beanbox" whose library APIs provide support for creating a variety of component-oriented applications. In particular, Sieve

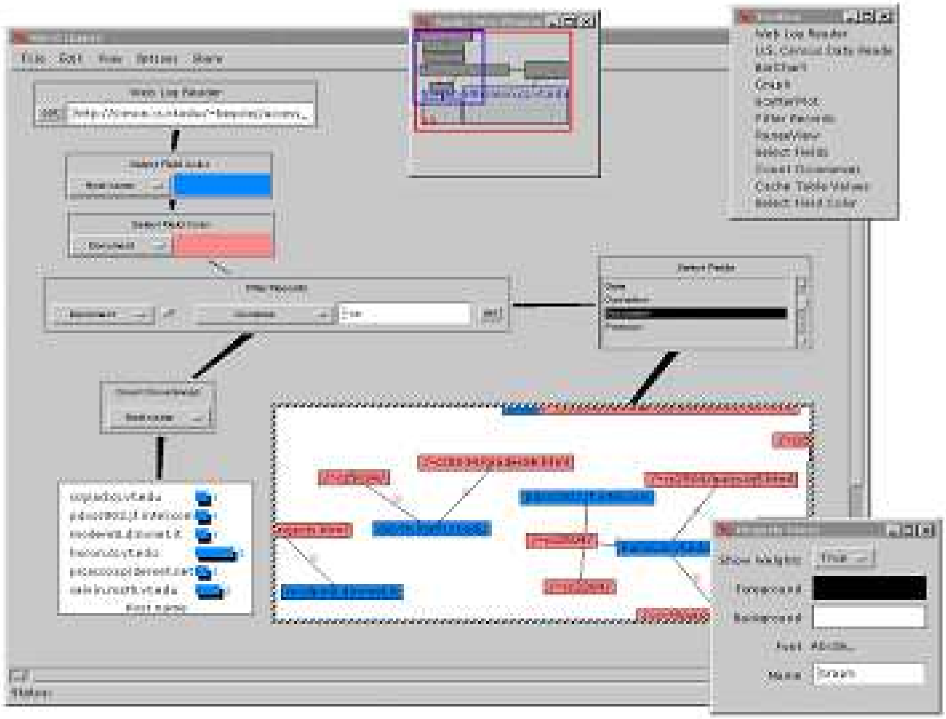


Figure 13: Example of a Sieve workspace with dataflow and annotations.

provides to programmers convenient facilities for defining not only the component themselves, but also for defining the actions performed by the links between components, along with specifying the format of the data that flows along those links. Together with Symphony (see next section), Sieve provides the foundation for constructing PSEs, as their combination creates a collaborative environment for controlling distributed, legacy resources.

Sieve provides an environment for collaborative component composition that supports the following:

- A Java-based system compatible with standard WWW browsers
- A convenient environment for generating visualizations through linking of data-producing modules with data-visualization modules
- Collaboration between users through a shared workspace, permitting all users to see the same visualizations at the same time
- Support for annotating the common workspace, visible to all users
- A convenient mechanism for linking in new types of components

Sieve presents the user with a large, scrollable workspace onto which data sources, processing modules, and visualization components may be dropped, linked, and edited. Figure 13 shows a Sieve workspace containing a simple data-flow network. This example illustrates a particular collection of beans that support various statistical visualizations.

Our design for Sieve allows processing and visualization modules to be generic, with all data-source–specific details hidden by the source modules. In addition, data-flow semantics can be tailored to the individual application characteristics and available grid services. For instance, the components of Figure 13 conform to an API which allows data to be viewed by adjacent modules in the network as a two-dimensional table containing objects of any type supported by the Java language. For other applications, the components could invoke grid services and file transfer utilities to effect the data-flow. Source modules simply convert raw data into the assumed data-flow representation. Processing modules can manipulate these data and present an altered or extended view. Visualization modules can then produce visual representations of the data in the system and also serve as interfaces for data selection.

Sieve's support for defining link types permits a great deal of flexibility in combining various collections of components. For example, if a component had been created that did not support the table format just described for our current visualization components, all that would be needed is for the programmer to create a link class that converts between the two data formats in question. The Sieve runtime environment will automatically deduce the appropriate link class to use when two components are linked together. Thus, if the new component (with its unique data format) were the source for the link, and one of the existing (table driven) visualization components were the target, then Sieve would determine that the link class to instantiate is the one that does the appropriate data conversion between the two components.

3.7 Symphony: Managing Remote Legacy Resources

Symphony is a component-based framework for composing, saving, sharing, and executing simulations [73]. Symphony views a simulation as a collection of programs and data resources intended to be executed as a unit. It is composed by assembling individual Symphony components into an acyclic directed graph and thus corresponds to a representation of a model or model instance as described in Section 1.4. The directed graph models the data flow relationships among the individual components. Such a graph is shown in Fig. 14 and described later. Each Symphony component is a surrogate for some actual program or data resource. The actual resources may, and usually do, reside on different machines that are able to communicate through an underlying network. In addition to modeling data-flow relationships, the connections between two Symphony components allow the components to synchronize the behavior of the computational resources they represent. Simulations may be saved by their creators for repeated reuse and shared with other users in a problem-solving community. Simulations are transparently executed in a manner that respects the data-flow requirements of individual programs in the composition. Execution transparency in this context means that all system level operations of program execution and of data movement across geographically distributed locations must be largely, if not totally, transparent to the user.

The Symphony framework, based on Sun's JavaBeans component architecture [49], has two principal elements: a composition environment in which a simulation is constructed and a back-end execution environment in which the described computation is performed. Symphony beans can be placed in any runtime environment that supports JavaBeans, such as Sieve (see Section 3.6). Fig. 14 shows Symphony running in Sun's Java BeanBox. The BeanBox interface allows generic Symphony components to be selected from a palette (not shown in Fig. 14) and dropped into the BeanBox's workspace. The properties of components in the workspace can be changed through customization to contain information about a specific computation resource for which the bean is a surrogate. Symphony includes Program beans that represent executable entities on some machine, and several beans for representing sources or destinations of data including a File bean, and beans that model input and output data streams.

The standardized architecture of JavaBeans provides for a large range of possible composition environments and operating systems that Symphony can work with. One such composition environment is Sieve (see Section 3.6), which adds the ability for several users to remotely collaborate on the same simulation. While Symphony was originally developed to exploit the graphical user interface of composition environments, we also envision that a simulation can be constructed and executed based on other representations (e.g., an XML description derived from a query-based specification of model instances). Languages for Grid workflow specification [11] are also relevant here.

When a simulation is executed, the components initiate, monitor, and synchronize the operations performed in a back-end execution environment. The back-end execution environment and the components collaborate to transport files between machines, execute programs, and connect data streams as needed to realize the computation specified in the network of beans. The operations are performed so as to respect the defined dataflow relationships defined by the simulation. The back-end execution environment was originally based on a specially created server. Our

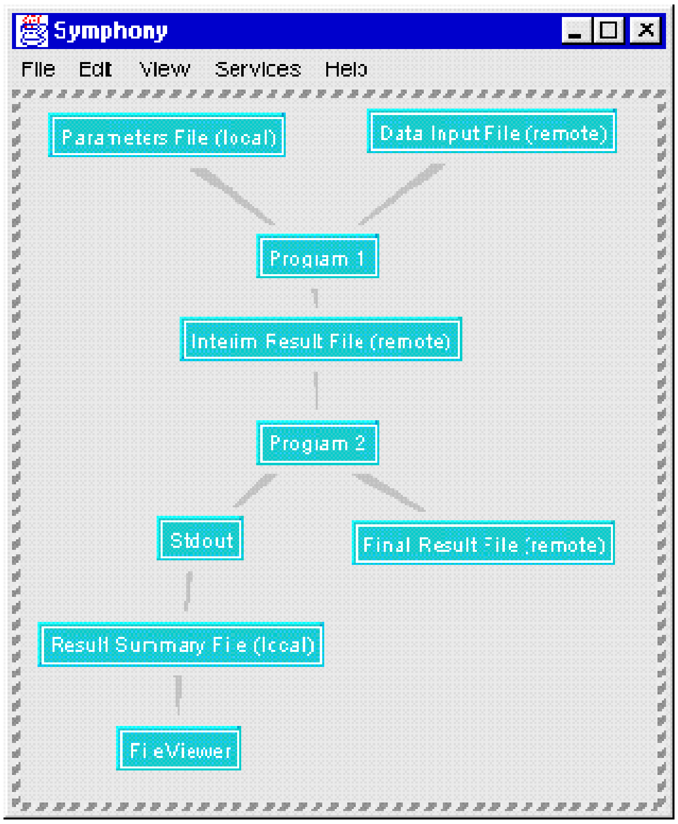


Figure 14: Design of a Symphony Bean Network.

current goal is to interface Symphony with one or more computational grids and networks of workstations for this purpose. In general, we envision Symphony as an abstraction layer on top of popular grid middleware systems that will ease and unify the access to grid resources and provide a foundation for collaboration among grid users through the exchange of pre-configured simulation components or even whole simulation. We believe that such a framework would be valuable to the community and would foster the more widespread application of computational grids significantly.

A simple simulation is depicted in Fig. 14. The simulation shown in Fig. 14 represents a remote program (Program 1) which takes a parameter file and a data file as input and generates an intermediate result file. This file is then used by another program (Program 2) to generate the final result which is also stored in a file and a summary of the computation which is sent to the standard output. The standard output in turn is redirected to a local file that is used as the input for a local file viewer application (a helper bean).

In terms of the functional layers presented in Fig. 2, Symphony provides both simulation definition and model definition services. Symphony provides the runtime structure to synchronize the execution of a collection of programs without user intervention. As noted above, a configured and connected set of Symphony beans could be generated by programmatic means or from some syntactic (e.g., XML) description. This description might itself be produced by the parameter definition layer in Fig. 2. However, Symphony currently lacks a number of important features that should be provided by simulation definition subsystems. For example, it should be possible to build cyclic simulations that would execute repeatedly. Such a simulation would include a component to determine termination based on convergence or other optimization criteria. Additionally, support for parameter sweep style execution should be provided. Exploiting the concurrent execution of programs unrelated through data-flow constraints is also desirable. Symphony currently provides some facilities at the model management layer because it, in cooperation with the composition environment, provide a user interface through which a simulation (a model) can be constructed. More sophisticated model definition features are clearly desired. For example, it should be possible to aggregate part or all of a simulation as a single unit. This would allow for easier organization of larger simulations and also provide a better basis for sharing complex assemblies of components among users.

Some of the issues we are currently addressing in our work with Symphony are: defining a general syntax for resource declaration and job specification, support for the Globus middleware [33] and it's security infrastructure, support for other middleware and security architectures, providing means for more effective data routing between resources, and support for resource discovery mechanisms and automatic resource allocation through super schedulers. A brief discussion follows.

A general syntax for resource declaration and job specification is needed to denote computational resources in a grid-independent manner. We have had initial success with implementing a resource description that lets us create program beans which abstract programs accessible through our local Globus grid by using the Globus Commodity Toolkit for Java. Our program beans can be customized with information collected from a Grid Information Index Service (GIIS [23]), from a local resource configuration file or through manually entered parameters. We also added a Globus authentication service which provides an interface for all Symphony beans to get access to a default Globus user proxy that these beans can use to authenticate themselves to grid resources.

We are currently investigating the use of more efficient methods than used in the original version of Symphony to transport data files as well as executables from one location to another. For the Globus interface we are looking at employing third-party GSI-FTP and standard FTP services as well as globus access to secondary storage (GASS [10]) services which are based on the HTTPS protocol.

We are examining resource discovery and automatic resource allocation through super schedulers. We propose to interface with a resource brokerage system which could automate the resource selection process for compute resources by comparing the constraints stated by the software resource, user preferences and user rights against to the properties of the available resources in the grid.

The Symphony framework requires a security architecture that will provide for finer grained and more flexible control of rights than is currently available on computational grids. The requirement for finer grained control arises from Symphony's ability to exchange simulation components and whole simulations between users. Current grid security infrastructures do not support these requirements. We believe that elements of the CRISIS [9] architecture may prove to be very useful to the aims of Symphony and should be implemented in future grid security systems.

4 Discussion

Our approach to GCEs can be characterized by an emphasis on high-level problem-solving facilities, and their implementation using traditional grid information and lower-level services, for multidisciplinary grid communities. Our unifying framework for multidisciplinary applications leverages concepts from data management, model representation, and compositional modeling into a cohesive methodology.

By viewing descriptions of model instances (and simulations) as entries in a database, programmatic descriptions of new model instances (and simulations) can be automatically created by writing queries. By writing BSML specifications and using Symphony to associate the resulting simulations with the underlying computational environment, scientists and engineers are able to interact with GCEs in the vernacular of the domain. The distinctions made by our framework mirror other projects such as CACTUS applications [5], where the emphasis is on creating portals that integrate large-scale simulation and visualization.

The presented techniques also allow us to store descriptions, 'run' the descriptions to obtain data, record the data back in the database, and associate the data with the description that corresponds to its experimental setup. This emphasis on the entire problem-solving context facilitates sophisticated services such as change management.

In contrast to the variety of standards (many, XML-based) available for scientific data, our data model is aimed at capturing representations of simulations, not just simulation data. We posit that the description of a simulation is a more persistent representation of the data (it produces) than the data itself. As technology matures and evolves, recording how specific data was obtained is important for the purposes of ensuring repeatability and reliability. For example, if gridding technology for microarrays improves, then 'running' the same (stored) description with the new setup can be used to arrive at new, current, results. Since there is nothing in our design that commits us to a rigid schema, our data model can elegantly adapt to changes over time.

These requirements point to the future directions in the development of GCEs. We are now extending our ideas to apply to runtime scenarios, such as computational steering and closing-the-loop between design and analysis (see Section 2.5). Runtime recommendation of models will become pertinent in heterogeneous and distributed scenarios, where information about application characteristics is acquired only during the computation. Connections to grid information services have to be established to enable GCEs to participate in large-scope projects that span multiple institutions and data sources.

The 'simulation as querying' viewpoint provides a useful conceptual abstraction for the effective utilization of computational grids. A long-term goal is to design GCEs that help unify modeling, simulation, analysis, and design activities — this is especially pertinent in multidisciplinary applications. Our architecture and systems support technologies are uniquely designed to support such an integrated mode of investigation.

Acknowledgements

We acknowledge the numerous collaborators, post-doctoral research associates, and students who have been involved in the Virginia Tech PSEs research group over the past several years. The work presented in this paper is supported in part by National Science Foundation grants DMS-9625968, EIA-9974956, EIA-9984317, and MCB-0083315, NASA grant NAG-2-1180, AFOSR grant F496320-99-1-0128, the Defense Advanced Research Projects Agency through the Office of Naval Research (N00014-01-1-0852), and the Virginia Tech ASPIRES program.

References

[1] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann, San Francisco, CA, 2000.

[2] V.S. Adve, R. Bagrodia, J.S. Browne, E. Deelman, A. Dube, E.N. Houstis, J.R. Rice, R. Sakellariou, D.J. Sundaram-Stukel, P.J. Teller, and M.K. Vernon. POEMS: End-to-End Performance Design of Large Parallel Adaptive Computational Systems. *IEEE Transactions on Software Engineering*, Vol. 26(11):pages 1027–1048, November 2000.

[3] V.S. Adve and R. Sakellariou. Application Representations for Multiparadigm Performance Modeling of Large-Scale Parallel Scientific Codes. *International Journal of High Performance Computing Applications*, Vol. 14(4):pages 304–316, Winter 2000.

[4] G. Allen, W. Benger, T. Goodale, H.-C. Hege, G. Lanfermann, A. Merzky, T. Radke, E. Seidel, and J. Shalf. Cactus Tools for Grid Applications. *Cluster Computing*, 2001. to appear.

[5] G. Allen, T. Goodale, G. Lanfermann, T. Radke, E. Seidel, W. Benger, H.-C. Hege, A. Merzky, J. Massó, and J. Shalf. Solving Einstein's Equations on Supercomputers. *IEEE Computer*, Vol. 32(12):pages 52–58, December 1999.

[6] R.G. Alscher, B.I. Chevone, L.S. Heath, and N. Ramakrishnan. Expresso - A PSE for Bioinformatics: Finding Answers with Microarray Technology. In A. Tentner, editor, *Proceedings of the High Performance Computing Symposium, Advanced Simulation Technologies Conference*, pages 64–69, April 2001.

[7] P. Avery and I. Foster. Petascale Virtual-Data Grids for Data Intensive Science. White paper accessible from URL: `http://www.griphyn.org`, April 2000.

[8] M. Baker, R. Buyya, and D. Laforenza. The Grid: International Efforts in Global Computing. In *Proceedings of the International Conference on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet (SSGRR 2000)*. Italy, 2000.

[9] E. Belani, A. Vahdat, T. Anderson, and M. Dahlin. The CRISIS Wide Area Security Architecture. In *Proceedings of the USENIX Security Symposium, San Antonio, TX*, pages 15–30, January 1998.

[10] J. Bester, I. Foster, C. Kesselman, J. Tedesco, and S. Tuecke. GASS: A Data Movement and Access Service for Wide Area Computing Systems. In *Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems*, May 1999.

[11] H.P. Bivens. Grid Workflow. Grid Computing Environments Working Group Document, 2001.

[12] F. Bodin, P. Beckman, D. Gannon, S. Narayana, and S.X. Yang. Distributed pC++: Basic Ideas for an Object Parallel Language. *Scientific Programming*, Vol. 2(3):pages 7–22, 1993.

[13] R. Bramley, K. Chiu, S. Diwan, D. Gannon, M. Govindaraju, N. Mukhi, B. Temko, and M. Yochuri. A Component Based Services Architecture for Building Distributed Applications. In *Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC'00)*. IEEE Press, 2000.

[14] J.C. Browne, E. Berger, and A. Dube. Compositional Development of Performance Models in POEMS. *International Journal of High Performance Computing Applications*, Vol. 14(4):pages 283–291, Winter 2000.

[15] H. Casanova and J. Dongarra. NetSolve: A Network Server for Solving Computational Science Problems. *International Journal of Supercomputer Applications and High Performance Computing*, Vol. 11(3):pages 212–223, Fall 1997.

[16] H. Casanova, J. Dongarra, C. Johnson, and M. Miller. Application-Specific Tools. In C. Kesselman and I. Foster, editors, *The Grid: Blueprint for a New Computing Infrastructure*, pages 159–180. Morgan Kaufmann, 1998.

[17] H. Casanova, G. Obertelli, F. Berman, and R. Wolski. The AppLeS Parameter Sweep Template: User-Level Middleware for the Grid. In *Proceedings of the Supercomputing Conference (SC'2000)*, 2000.

[18] K.M. Chandy, R. Bramley, B.W. Char, and J.V.W. Reynders. Report of the NSF Workshop on Problem Solving Environments and Scientific IDEs for Knowledge, Information and Computing (SIDEKIC'98). Technical report, Los Alamos National Laboratory, 1998.

[19] K.M. Chandy and C. Kesselman. *CC++: A Declarative, Concurrent, Object Oriented Programming Notation*. Research Directions in Concurrent Object-Oriented Programming. MIT Press, 1993.

[20] S. Chawathe and H. Garcia-Molina. Meaningful Change Detection in Structured Data. In *Proceedings of the ACM-SIGMOD Conference on Management of Data*, pages 26–37. Tucson, Arizona, USA, 1997.

[21] A. Cleary, S. Kohn, S.G. Smith, and B. Smolinski. Language Interoperability Mechanisms for High-Performance Scientific Applications. Technical Report UCRL-JC-131823, LLNL, 1998.

[22] J.C. Cummings, J.A. Crotinger, S.W. Haney, W.F. Humphrey, S.R. Karmesin, J.V.W. Reynders, S.A. Smith, and T.J. Williams. Rapid Application Development and Enhanced Code Interoperability using the POOMA framework. In *Proceedings of the SIAM Workshop on Object-Oriented Methods and Code Interoperability in Scientific and Engineering Computing (OO'98)*. SIAM Press, 1998.

[23] S. Czajkowski, K. Fitzgerald, I. Foster, and C. Kesselman. Grid Information Services for Distributed Resource Sharing. In *Proceedings of the Tenth IEEE International Symposium on High-Performance Distributed Computing*. IEEE Press, August 2001.

[24] K.M. Decker and B.J.N. Wylie. Software Tools for Scalable Multilevel Application Engineering. *International Journal of High Performance Computing Applications*, Vol. 11(3):pages 236–250, Fall 1997.

[25] T.T. Drashansky. *An Agent Based Approach to Multidisciplinary Problem Solving Environments*. PhD thesis, Dept. of Computer Sciences, Purdue University, 1996.

[26] T.T. Drashansky, E.N. Houstis, N. Ramakrishnan, and J.R. Rice. Networked Agents for Scientific Computing. *Communications of the ACM*, Vol. 42(3):pages 48–54, March 1999.

[27] T.M. Eidson. A Component-Based Programming Model for Composite, Distributed Applications. Technical Report ICASE Report No. 2001-15; NASA/CR-2001-21087, Institute for Computer Applications in Science and Engineering (ICASE), May 2001.

[28] B. Falkenhainer and K. Forbus. Compositional Modeling: Finding the Right Model for the Right Job. *Artificial Intelligence*, Vol. 51:pages 95–143, 1991.

[29] P.A. Fishwick. *Simulation Model Design and Execution: Building Digital Worlds*. Prentice Hall, 1995.

[30] K.D. Forbus. Qualitative Reasoning. In A.B. Tucker, editor, *The Computer Science and Engineering Handbook*, pages 715–733. CRC Press, 1996.

[31] I. Foster. Compositional Parallel Programming Languages. *ACM Transactions on Programming Languages and Systems*, Vol. 18(4):pages 454–476, July 1996.

[32] I. Foster, J. Insley, G. Von Laszewski, C. Kesselman, and M. Thiebaux. Distance Visualization: Data Exploration on the Grid. *IEEE Computer*, Vol. 32(12):pages 36–43, December 1999.

[33] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications and High Performance Computing*, Vol. 11(2):pages 115–128, Summer 1997.

[34] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, July 1998.

[35] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of Supercomputer Applications*, 2001. to appear.

[36] G. Fox, D. Gannon, and M. Thomas. Grid Computing Environments Working Group Specification. Global Grid Forum, 2001.

[37] G.C. Fox and W. Furmanski. High-Performance Commodity Computing. In C. Kesselman and I. Foster, editors, *The Grid: Blueprint for a New Computing Infrastructure*, pages 237–255. Morgan Kaufmann, 1998.

[38] D. Gannon. Grid Information Services and XML Schemata. White paper accessible from URL: `http://www.extreme.indiana.edu/~gannon/schemata.html`, 2000.

[39] D. Gannon, B. Bramley, T. Stuckey, J. Villacis, J. Balasubramanian, E. Akman, F. Breg, S. Diwan, and M. Govindaraju. Component Architectures for Distributed Scientific Problem Solving. *IEEE Computational Science and Engineering*, Vol. 5(2):pages 50–63, May/June 1998.

[40] D. Gannon, R. Bramley, T. Stuckey, J. Villacis, J. Balasubramanian, E. Akman, F. Breg, S. Diwan, and M. Govindaraju. The Linear System Analyzer. In E.N. Houstis, J.R. Rice, E. Gallopoulos, and R. Bramley, editors, *Enabling Technologies for Computational Science*, pages 123–134. Kluwer Academic Publishers, 2000.

[41] D. Gannon and A. Grimshaw. Object-Based Approaches. In C. Kesselman and I. Foster, editors, *The Grid: Blueprint for a New Computing Infrastructure*, pages 205–236. Morgan Kaufmann, 1998.

[42] A. Goel, C.A. Baker, C.A. Shaffer, B. Grossman, W.H. Mason, L.T. Watson, and R.T. Haftka. VizCraft: A Problem-Solving Environment for Aircraft Configuration Design. *IEEE/AIP Computing in Science and Engineering*, Vol. 3(1):pages 56–66, 2001.

[43] A. Goel, C. Phanouriou, F.A. Kamke, C.J. Ribbens, C.A. Shaffer, and L.T. Watson. WBCSim: A Prototype Problem Solving Environment for Wood-Based Composites Simulation. *Engineering with Computers*, Vol. 15:pages 198–210, 1999.

[44] R.L. Grossman, D. Hanley, and S. Bailey. High Performance Data Management: A Case for Using Lightweight, High Performance Persistent Object Managers in Scientific Computing. Technical Report 95–R18, Laboratory for Advanced Computing, University of Illionis, Chicago, 1995.

[45] H. Hamadeh and C.A. Afshari. Gene Chips and Functional Genomics. *American Scientist*, Vol. 88:pages 508–515, November-December 2000.

[46] L.S. Heath, N. Ramakrishnan, R.R. Sederoff, R.W. Whetten, B.I. Chevone, C.A. Struble, V.Y. Jouenne, D. Chen, L. van Zyl, and R.G. Alscher. Modeling with the Expresso Microarray Experiment Management System: The Functional Genomics of Stress Responses in Loblolly Pine. *Genome Research*, 2001. Communicated for publication.

[47] J.M. Hellerstein. Optimization Techniques for Queries with Expensive Methods. *ACM Transactions on Database Systems*, Vol. 23(2):pages 113–157, September 1998.

[48] E.N. Houstis, A.C. Catlin, J.R. Rice, V.S. Verykios, N. Ramakrishnan, and C.E. Houstis. PYTHIA-II: A Knowledge/Database System for Managing Performance Data and Recommending Scientific Software. *ACM Transactions on Mathematical Software*, Vol. 26(2):pages 227–253, June 2000.

[49] Sun Microsystems Inc. The JavaBeans Component Architecture. White paper accessible from URL: `http://java.sun.com/products/javabeans/`, 1998.

[50] Y. Ioannidis, M. Livny, S. Gupta, and N. Ponnekanti. ZOO: A Desktop Experiment Management Environment. In *Proceedings of the Twenty Second International Conference on Very Large Databases (VLDB'96)*, pages 274–285, 1996.

[51] J. Jiang, K.K. Bae, W.H. Tranter, A. Verstak, N. Ramakrishnan, J. He, L.T. Watson, T.S. Rappaport, and C.A. Shaffer. S^4W: A Collaborative PSE for Modeling of Broadband Wireless Communication Systems. In *Proceedings of the Eleventh Annual MPRG Symposium on Wireless Personal Communications*, 2001.

[52] C. Johnson, S.G. Parker, C. Hansen, G.L. Kindlmann, and Y. Livnat. Interactive Simulation and Visualization. *IEEE Computer*, Vol. 32(12):pages 59–65, December 1999.

[53] S. Karin and S. Graham. The High Performance Computing Continuum. *Communications of the ACM*, Vol. 41(11):pages 32–35, November 1998.

[54] K. Keahey, P. Beckman, and J. Ahrens. Ligature: Component Architecture for High Performance Applications. *International Journal of High Performance Computing Applications*, Vol. 14(4):pages 347–356, Winter 2000.

[55] K. Kennedy. Telescoping Languages: A Compiler Strategy for Implementation of High-Level Domain-Specific Programming Systems. In *Proceedings of the 14th International Parallel and Distributed Processing Symposium (IPDPS'00)*, 2000.

[56] D.L. Knill, A.A. Giunta, C.A. Baker, B. Grossman, W.H. Mason, R.T. Haftka, and L.T. Watson. Response Surface Models Combining Linear and Euler Aerodynamics for Supersonic Transport Design. *Journal of Aircraft*, Vol. 36(1):pages 75–86, 1999.

[57] K.R. Koch, R.S. Baker, and R.E. Alcouffe. Solution of the First-Order Form of the 3-D Discrete Ordinates Equation on a Massively Parallel Processor. *Transactions of the American Nuclear Society*, Vol. 65(198), 1992.

[58] B. Kuipers. Qualitative Simulation. *Artificial Intelligence*, Vol. 29:pages 289–338, 1986.

[59] Lawrence Livermore National Laboratory. The ASCI Sweep3D Benchmark Code. Accessible from URL: `http://www.llnl.gov/asci_benchmarks`, 1995.

[60] R.W. Moore, C. Baru, R. Marciano, A. Rajasekar, and M. Wan. Data-Intensive Computing. In C. Kesselman and I. Foster, editors, *The Grid: Blueprint for a New Computing Infrastructure*, pages 105–129. Morgan Kaufmann, 1998.

[61] R.W. Moore, T.A. Prince, and M. Ellisman. Data-Intensive Computing and Digital Libraries. *Communications of the ACM*, Vol. 41(11):pages 56–62, November 1998.

[62] P.P. Nayak. *Automated Modeling of Physical Systems*. PhD thesis, Stanford University, 1992.

[63] N. Ramakrishnan. Experiences with an Algorithm Recommender System. In P. Baudisch, editor, *Working Notes of the CHI'99 Workshop on Interacting with Recommender Systems*. ACM SIGHI Press, 1999.

[64] N. Ramakrishnan and A.Y. Grama. Mining Scientific Data. *Advances in Computers*, Vol. 55:pages 119–169, 2001.

[65] N. Ramakrishnan, E.N. Houstis, and J.R. Rice. Recommender Systems for Problem Solving Environments. In H. Kautz, editor, *Technical Report WS-98-08 (Working Notes of the AAAI-98 Workshop on Recommender Systems)*, pages 91–95. AAAI/MIT Press, 1998.

[66] N. Ramakrishnan and C.J. Ribbens. Mining and Visualizing Recommendation Spaces for Elliptic PDEs with Continuous Attributes. *ACM Transactions on Mathematical Software*, Vol. 26(2):pages 254–273, June 2000.

[67] N. Ramakrishnan and R.E. Valdés-Pérez. Note on Generalization in Experimental Algorithmics. *ACM Transactions on Mathematical Software*, Vol. 26(4):pages 568–580, December 2000.

[68] J.R. Rice and R.F. Boisvert. From Scientific Software Libraries to Problem-Solving Environments. *IEEE Computational Science & Engineering*, Vol. 3(3):pages 44–53, Fall 1996.

[69] J. Rickel and B. Porter. Automated Modeling of Complex Systems for Answering Prediction Questions. *Artificial Intelligence*, Vol. 93:pages 201–260, 1997.

[70] E.J. Rubin, R. Dietz, J. Chanat, C. Speir, R. Dymond, V. Lohani, D. Kibler, D. Bosch, C.A. Shaffer, N. Ramakrishnan, and L.T. Watson. From Landscapes to Waterscapes: A PSE for Landuse Change Analysis. In M. Deville and R. Owens, editors, *Proceedings of the 16th IMACS World Congress*, August 2000.

[71] J. Saltz, A. Sussman, S. Graham, J. Demmel, S. Baden, and J. Dongarra. Programming Tools and Environments. *Communications of the ACM*, Vol. 41(11):pages 64–73, November 1998.

[72] E. Shaffer, D.A. Reed, S. Whitmore, and B. Schaeffer. Virtue: Performance Visualization of Parallel and Distributed Applications. *IEEE Computer*, Vol. 32(12):pages 44–51, December 1999.

[73] A. Shah and D.G. Kafura. Symphony: A Java-based Composition and Manipulation Framework for Distributed Legacy Resources. In *Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems*, pages 2–12. Los Angeles, CA, May 1999.

[74] M. Stonebraker. Sequoia 2000: A Reflection on the First Three Years. *IEEE Computational Science and Engineering*, Vol. 1(4):pages 63–72, Winter 1994.

[75] A. Szalay, P. Kunszt, A. Thakar, and A. Gray. Designing and Mining Multi-Terabyte Astronomy Archives: The Sloan Digital Sky Survey. In *Proceedings of the ACM-SIGMOD Conference on Management of Data*, pages 451–462, 2000.

[76] C. Upson, T. Faulhaber, D. Kamins, D. Schlegel, D. Laidlaw, F. Vroom, R. Gurwitz, and A. van Dam. The Application Visualization System: A Computational Environment for Scientific Visualization. *IEEE Computer Graphics and Applications*, Vol. 9(4):pages 30–42, 1989.

[77] S. Verma, M. Parashar, G. von Laszewski, and J. Gawor. Corba Community Grid Toolkit (CoG). Grid Computing Environments Community Practice (CP) Document, 2001.

[78] A. Verstak, M. Vass, N. Ramakrishnan, C.A. Shaffer, L.T. Watson, K.K. Bae, J. Jiang, W.H. Tranter, and T.S. Rappaport. Lightweight Data Management for Compositional Modeling in Problem Solving Environments. In A. Tentner, editor, *Proceedings of the High Performance Computing Symposium, Advanced Simulation Technologies Conference*, pages 148–153, April 2001.

[79] G. von Laszewski. The Grid Portal Development Kit. Grid Portal Collaboration Effort Document, 2001.

[80] M. Young, D. Argiro, and S. Kubica. Cantata: Visual Programming Environment for the Khoros System. *Computer Graphics*, Vol. 29(2):pages 25–28, 1995.