DLR's VirtualLab: Scientific Software Just a Mouse Click Away

By Thilo Ernst, Tom Rother, Franz Schreier, Jochen Wauer, and Wolfgang Balzer

THE INTERNET HAS BECOME THE BACKBONE OF SCIENTIFIC INFORMATION EXCHANGE, WITH EMAIL BEING THE PRIMARY COMMUNICATION MEDIUM FOR SCIENTISTS. SCIENTIFIC PAPERS AND REPORTS ARE USUALLY OFFERED ON UNIVERSITY AND

research institute Web servers and indexed by Web-accessible databases. Many major journals have an online edition—some are even available electronically only.

Scientific software, too, is exchanged routinely over the Internet, with many research organizations offering software developed in the course of their work for download on Web or ftp servers.[1] However, accessing such software is not as easy as accessing scientific papers. When you click on a link to a paper in your Web browser, for example, the document is downloaded and displayed immediately; intermediate steps such as the activation of a browser plug-in occur automatically behind the scenes. Unfortunately, software authors in the research sector can't port their programs to a variety of platforms and package them into convenient automatic installers. Users must learn on their own how to install downloaded programs and make them run on their platforms.

Even if the downloaded software's installation presents no problems for users, inadequate documentation can make performing productive work with that software impossible. Retrieving suitable programs for a given problem domain is also much harder because the programs themselves aren't indexed (although dedicated databases[2] alleviate this problem to a certain extent). The best bet is to pursue software described in indexed publications that mention a download address, but such publications are usually theoretical papers that don't provide sufficient detail on how to actually use the program.

All these factors likely contribute to a certain psychological resistance to trying out other people's scientific programs, even if these programs are readily available for download: seeing the cost–benefit ratio in advance is just too hard. Basically, the Internet revolution has not brought about quite the same level of ubiquity and smooth access to scientific software that scientific documents have enjoyed. To improve this situation, we propose virtual labs—Web-based searchable repositories run by research organizations that make broad collections of scientific software available for online use. Program execution occurs on the server side, so users are freed completely from installation and porting issues. They only need a Web browser and Internet connectivity to retrieve and use programs, which isn't any more than required to access a wealth of scientific documents today.

The Virtual Lab Concept

The term *virtual laboratory* has been in use for some years on projects emphasizing a diverse set of themes—for example, Web-based remote access to physical laboratory equipment (www.csm.ornl.gov/newVL.html), desktop or Web-based e-learning multimedia packages for different scientific disciplines (www.hhmi.org/biointeractive/vlabs/index.htm and www.vlab.com), and Web-based collections of educational Java applets (www.phy.ntnu.edu.tw/java/index.html). Only recently have virtual labs emerged that focus on providing Web-based access to scientific software, albeit for limited specific problem areas and target audiences. Examples include the National Institute of Standards and Technology's Virtual Cement and Concrete Testing Laboratory (www.bfrl.nist.gov/861/vcctl) and Web platforms for entire classes of applications (such as PISE, the Pasteur Institute Software Environment for molecular biology software,[3] which inspired some key features of our system).

Unlike related approaches, the virtual lab concept targets the broad spectrum of real scientific software that is the bread and butter of research organizations today. By letting them easily meet on the Web, the VirtualLab we developed at the German Aerospace Center (DLR) aims to bring scientific programs and their potential users closer (see Figure 1). Our VL concept has a distinctive, three-part goal: to make broad scientific applications available for online execution via the Inter-

Dave's Sideshow

The Hunt for the Elusive "Mondo"

One of the perks of being a professor is giving students projects in which the specifications are slightly ambiguous, misleading, or simply unstated. Case in point: the threads library project I recently gave my operating systems class. The key to any "good" project is to quietly add some sort of mysterious requirement—in this case, a magic `THREAD_MONDO` flag that may be optionally passed to a function that is supposed to create a thread. Obviously, this leads to questions: "What is this Mondo of which you speak?" "Was the Mondo property covered in the class I missed?" "Can Mondo and non-Mondo threads share state?"

The very presence of "Mondo" has led more imaginative students to engage in wild speculation and conjecture about the interpretation of `THREAD_MONDO` and its possible relationship to mondo-threads, mondo-context, mondo-locks, mondo-inversion, mondo-deadlock, mondo-priority, mondo-scheduling, mondo-inheritance, meta-mondo, mondo-ware, mondo-diffusion, mondo-exclusion, mondo-flops, mondo-grids, and my personal favorite, the mondo uncertainty principle (which states that the simultaneous specification and implementation of "Mondo" can never be achieved with absolute certainty). Needless to say, Mondo is much more complicated than I ever imagined.

So, are we any closer to understanding Mondo? Who knows? However, I am sure that a federally funded multi-institutional collaborative Mondo research project would definitely help pin it down.

"You Put a Web Server Where?"

On the subject of Mondo, I seem to spend more than my fair share of time debugging. Like many programmers, I freely admit that my debugging tool of choice tends to be the "print" statement. This works just fine for simple things, and it's easy enough to use. The problem is that there are many situations in which "print" really doesn't work so well. For instance, if you have a code with complicated data structures such as trees and graphs, figuring out what you're looking at can be very difficult. Similarly, I have fond memories of debugging a molecular dynamics code on the Connection Machine where inserting a few innocent "print" statements suddenly resulted in several hundred megabytes of unintelligible debugging output.

To deal with complicated situations like this, I have turned to a new approach—debugging through the Web. Yes, that's right, the Web. The idea for this goes back a few years to when a co-worker at Los Alamos asked me if there were some simple way to monitor and interact with long-running simulations from home. Thinking about this for a bit, we realized that we could just build a simple Web server in Python (the control language for our application) and use that to provide remote access. This worked remarkably well—it only required an afternoon of coding, didn't require any difficult programming (mostly just print statements and HTML), and had the advantage of working from any machine that could browse the Web. It was cool.

Since then, we've played around with this Web server idea as a little side-project. Recently, a student and I decided to repackage the server as a C library that could simply be linked into an application and used to provide Web access with about the same simplicity as using print. More importantly, we have maintained the Internet tradition of naming the system after a liquid by calling the library "SWILL." Here is a simple SWILL example:

```c
#include "swill.h"

/* This function generates a web page */
void hello(FILE *f) {
    printf(f, "Hello World\n");
    }

int main() {
/* Open server on port 8080 */
swill_init(8080);

    /* Add a web-page */
    swill_handle("hello.html", hello, 0);
    while (1) {
        swill_serve();
        }
    }
```

Although the idea of using a Web server as an embedded library might sound weird, it has proven to be a pretty useful debugging tool. For instance, data structures such as trees and graphs are easily mapped to collections of Web pages where HTML "links" merely point to other parts of the data structure. Similarly, an embedded server can provide interactive access to large amounts of program state without having to suspend the application and create a huge debugging file. In my own work, I've used SWILL to help me debug compiler parse trees. I've also embedded it inside an operating system simulator so I could have real-time monitoring of machine state. It's really a lot of fun. If this sounds interesting, you can find further details at http://systems.cs.uchicago.edu/swill.

Until next time, it's time to get back to thinking of the next obscure project requirement.
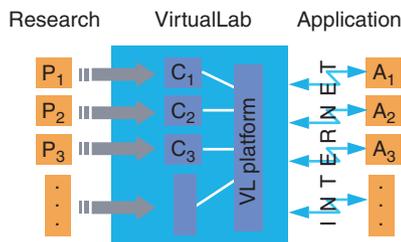
Figure 1. Bridging the gap. P represents scientific programs, C represents VirtualLab components, and A represents application problems.

net along with tightly integrated, searchable, coarsely standardized documentation under minimal effort.

Scientists tend to build general models even when targeting a specific context. For example, scattering and radiative transfer models are decisive elements in atmospheric remote-sensing applications, but they're also effective tools in technical and medical diagnostics. Mathematical modeling and computer-based simulation are routine tools in most scientific disciplines, thus scientific programs as executable implementations of models represent an important part of the "output" of research organizations.

Most models have a certain degree of generality, so researchers can often use the programs in contexts that differ from their "home" projects. However, the potential in doing so is rarely exploited—offering scientific software for Web-based online execution can change this. For example, making scientific programs from different research areas routinely accessible on the Web and thus substantially lowering the barrier to entry for new users will help them identify and exploit such application opportunities and thus foster technology transfer and scientific ex-

change. Of course, the concept will only succeed if a prospective user has a convenient way to learn about available programs in the first place and if there is enough (easily accessible) information about how to use them.

Likewise, the concept critically depends on the willingness of authors to integrate their programs into the platform—doing so must be very easy. The goal is *blackbox integration*: the program runs as is, with no internal changes whatsoever; the *integrator* (the person preparing the program for use in the VL) only needs to provide a certain amount of accompanying additional information.

Project Scope

The VL project's initial goals were to provide Web access for electromagnetic scattering and radiative transfer simulation applications developed at the DLR's Remote Sensing Technology Institute and to make them more accessible for technology transfer and scientific exchange. To reduce per-program development effort, it became clear early on that providing Web interfaces for a substantial selection of programs only would be feasible with a generic platform. No suitable platform was readily available, so we set out to build our own. The concept proved attractive to researchers from other disciplines as well; the constraints we placed on the design were technically motivated rather than connected to our particular research area.

We identified noninteractive (or pseudo-interactive) command-line applications as a useful application class. These programs can be written in different programming languages, run on different operating system and hardware platforms, and use arbitrary proprietary file formats. However, they all can be in-

voked in the same manner if the underlying operating system offers a minimal amount of Posix conformance. (Recently, we extended the project's scope to also cover an interesting class of graphical user interface applications.)

Because our concept involves concentrating all the computational resources needed on the server side, there must be limits on the resource demands of the programs to be integrated. We focus on software that can run on desktop systems or small clusters. Supercomputer or grid computing power currently is outside of our scope. Likewise, very complex scientific programs that have matured into products (for instance, the computational fluid dynamics codes used in industrial aerospace design) aren't in our focus. The interface complexity of such programs would likely exceed what a generic solution like the VL can provide. Moreover, they tend to be used routinely rather than occasionally, thus Web-based user interfaces seem suboptimal.

Functional Overview

We designed and implemented a VL prototype platform with several key features.

Unified Web Interfaces for Input, Execution, and Output

All scientific components provide coarsely unified (Web) user interfaces supporting data input, execution steering, and output. (Because the VL is a platform into which individual scientific programs are integrated, we refer to the integrated programs as *components*. This use of the term is somewhat sloppy because the VL does not constitute a component architecture in the strict sense of word; for instance, our interface descriptions do not support static type-checking.) The input Web user inter-

faces are dynamically generated from abstract interface descriptions.

Integrated Access to Documentation

The documentation (via hyperlinks from the Web user interface) provides context-sensitive online help, so during work with a program, the user can easily access that program's documentation and go directly to the location most appropriate to his or her current activity.

Component Database Search

A user can retrieve components based on search keywords matched against metadata that are part of the component documentation. Similar to a standard Internet search engine, the search result page provides overview information, which lets the user judge whether the component is interesting enough to explore further.

Personalized Data Management and Basic Security

Each user has a personal user area where all data resulting from work with the VL is stored along with the selection of components the user is working with and all experiment (simulation run) data. This private area is not accessible by other users or the public.

Integration Support and Documentation Management

Authors willing to integrate their software into the VL platform can use auxiliary programs to help them provide the additional information required, notably the coarsely standardized documentation format the VL uses.

Administration Support

Administrator Web pages and auxiliary programs support administrative activities such as user management and resource monitoring.
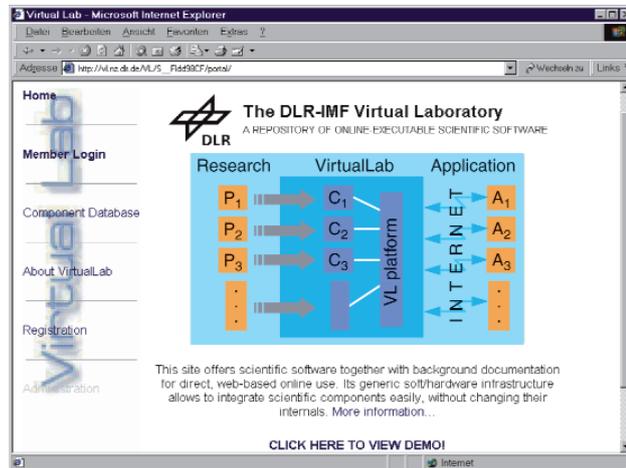

Figure 2. The German Aerospace Center's VirtualLab start page. This pilot system currently offers applications from two fields: scattering and radiative transfer.

Pilot System and Status of Implementation

Together with a selection of scientific components developed at the DLR's Remote Sensing Technology Institute and its partners, the VL platform prototype forms our pilot system—the Virtual Scattering and Radiative Transfer Lab (VSRL)—which is accessible at http://vl.nz.dlr.de (see Figure 2). A heterogeneous cluster containing seven Intel/Linux and one Sparc/Solaris machine is dedicated to this service. (See the "Applications in the Virtual Scattering and Radiative Transfer Lab" sidebar for further details about the VSRL.)

The component database is publicly accessible, but online execution of programs requires registration and is currently by invitation only (contact virtual-lab-nz@dlr.de). A movie-like demo of a user session is also openly available on the Web site.

The VL platform software will be made available to research organizations or other interested parties in the near future, but the detailed licensing terms are subject to ongoing negotia-

tions within DLR (send inquiries to virtual-lab-nz@dlr.de).

The VL Platform's Architecture

To not lock out users running old browsers, we based the project on first- and second-generation Web technology—mainly HTML 3.2 and HTTP 1.0. Today, developers are adopting "Web services" technologies and standards such as XML, SOAP, UDDI, and WSDL (www.w3.org) more widely. Such technologies might appear to be a natural choice for a project such as the VL, but at the time of the project's inception (mid 2000), they were largely on the horizon. Relying on them would have made development depend on moving targets. We also had to avoid discouraging conservative authors—requiring all documentations to be supplied in an XML format would have been a lot to ask, for example. The main implementation language we used is Python (www.python.org), an object-oriented dynamic language well suited for Web

Applications in the Virtual Scattering and Radiative Transfer Lab

Remote sensing of the Earth and its environment becomes more important in the context of sustainable development and global climate-change monitoring. Common to all remote-sensing techniques is the fact that they are based on information coming from scattered and transmitted electromagnetic waves in different spectral ranges. Therefore, realistic models of how such electromagnetic waves are transmitted and scattered under certain conditions are of particular importance in remote-sensing applications.

In the past, we developed several programs allowing light-scattering analysis on dielectric, but generally nonspherical, particles in the resonance region. In this region, the scattering models must be based on a full-wave analysis of Maxwell's equations—a mathematically challenging task. We also developed models to perform simulations in the infrared (line-by-line model) as well as the visible and ultraviolet spectral ranges. All these programs have been successfully applied in the course of various research projects to retrieve trace gases in the Earth's atmosphere, determine cloud and aerosol properties, and perform system studies for the detection of natural and anthropogenic high-temperature events (such as volcanoes and biomass burning).

Scattering Codes

In the VSRL, sophisticated programs allow light-scattering analysis up to the geometric optics region on various classes of nonspherical particles such as spheroids, hexagonal and irregular ice particles, and Chebyshev-like particles. (Researchers use the latter in remote sensing to model aerosol components, the microphysical properties of Cirrus clouds, and hydrometeors.) These programs include

- *Mieschka*, which calculates the integral and differential scattering properties of axisymmetric particles in a fixed orientation. It treats spherical particles as a special case.
- *Pmieschka*, which calculates the integral and differential scattering properties of randomly oriented axisymmetric particles. It also treats spherical particles as a special case.
- *CYL*, which computes integral and differential scattering quantities of finitely extended cylinders with circular, hexagonal, or octagonal cross sections in random orientation.
- *QCACP*, which models an inhomogeneous host scatterer that contains up to three different classes of small, densely packed spheres. A homogeneous analog with an effective permitivity replaces the inhomogeneous host scatterer. The method is based on the so-called "quasi-crystalline approach with coherent potential."[1]

The methodological background of mieschka, pmieschka, and CYL is the generalization of the separation of variables method.[2] In CYL, this method is applied in cylindrical coordinates in combination with Huygen's principle to find an approximation for finite cylindrical columns that have noncircular cross sections.[3] Essential numerical simplifications are achievable if the scattering geometry exhibits a certain symmetry.[4]

---

development and integration tasks involving scientific software.

Subsystem Structure

The VL platform's substructures closely correspond to its main tasks—to enable online retrieval and execution of scientific components, provide support for integrating new components, and support administrative tasks. The VL implementation is based on (and would not have been feasible without) powerful and mature open-source software.

Application Server

The application server accepts user requests, invokes actions from other subsystems, and transforms the results into a form consumable by the user's browser. The VL uses the ZOPE application server (www.zope.org), which is based on a unique object-based model, offers many community contributed extensions, and is implemented in Python, a feature that eases its interoperability with other subsystems.

Databases

In a bit of simplification, Figure 3 shows a single database. In reality, the VL platform uses four persistence mechanisms:

- The server file system stores the component executables along with the static data files that they require in order to function properly. Low-level experiment-specific data is stored in task directories.
- The ZOPE object database (ZODB) stores user-specific server areas (each user has a program and an experiment area) that contain all high-level experiment data. It also stores the VL portal Web site's static content.
- A Structured Query Language database (www.mysql.com) stores all component-specific metadata such as author name, title, and abstract.
- A lightweight directory access protocol directory (www.openldap.org) stores all registered users' credentials.

Experiment Configuration, Task Management, and Hardware Cluster

Each time a user (after supplying all his

Radiative Transfer Codes

Modeling the transfer of electromagnetic radiation in the atmosphere is important for meteorology, climatology, and atmospheric remote sensing. Absorption, emission, and the scattering of light at molecules, aerosols, and hydrometeors are the driving mechanisms of radiative transfer in the atmosphere.[5] The change of radiation intensity (radiance) passing through the atmosphere is formally described by the equation of radiative transfer, which cannot be solved analytically (except for trivial cases). Researchers have developed various radiative transfer codes in the past decades based on approximations appropriate for a certain application or wave number regime.

The following models are currently available in the VSRL for radiative transfer:

- *MIRART*: The *m*odules for *i*nfrared *a*tmospheric *r*adiative *t*ransfer are a suite of programs for high-resolution, line-by-line, radiative transfer calculations emphasizing efficient and reliable numerical algorithms and a modular approach appropriate for simulation and retrieval in atmospheric spectroscopy.[6] Recently, the Pyfort tool (http://pyfortran.sourceforge.net) has made individual MIRART modules accessible from a Python layer. These Python scripts are usually called from a Unix command line but are also available through the VL.

- *PFUI*: An earlier *P*ython *F*ascode *u*ser *i*nterface[7] was an inspiration for the VL's development. PFUI already significantly simplified the usage of the Fascode (fast atmospheric signature code), a line-by-line program that the US Air Force developed.[8] Now the same functionality is available in the VL's unified platform. (Due to licensing issues, only input-file generation is supported.)

References

1. L. Tsang, J.A. Kong, and R.T. Shin, *Theory of Microwave Remote Sensing*, John Wiley & Sons, New York, 1985.
2. T. Rother, "Generalization of the Separation of Variables Method for Nonspherical Scattering on Dielectric Objects," *J. Quantitative Spectroscopy & Radiative Transfer*, vol. 60, no. 3, 1998, pp. 335–353.
3. T. Rother, S. Havemann, and K. Schmidt, "Scattering of Plane Waves on Finite Cylinders with Noncircular Cross Sections," *Progress in Electromagnetic Research*, vol. 23, J.A. Kong, ed., EMW Publishing, Cambridge, Mass., 1999, pp. 79–105.
4. T. Rother, K. Schmidt, and S. Havemann, "Light Scattering on Hexagonal Ice Columns," *J. Optical Soc. Am. A*, vol. 18, no. 10, 2001, pp. 2512–2517.
5. K.-N. Liou, *An Introduction to Atmospheric Radiation*, Academic Press, San Diego, Calif., 1980.
6. F. Schreier and B. Schimpf, "A New Efficient Line-by-Line Code for High-Resolution Atmospheric Radiation Computations Including Derivatives," *IRS 2000: Current Problems in Atmospheric Radiation*, W.L. Smith and Y.M. Timofeyev, eds., A. Deepak Publishing, Hampton, Va., 2001, pp. 381–384.
7. F. Schreier and B. Schimpf, "PFUI—Python Fascode User Interface," *Atmospheric Millimeter and Submillimeter Wave Radiative Transfer Modeling II*, P. Eriksson and S. Bühler, eds., Logos–Verlag, Berlin, 2001, pp. 191–205.
8. F.X. Clough et al., "FASCOD3: Spectral Simulation," *IRS'88: Current Problems in Atmospheric Radiation*, J. Lenoble and J.F. Geleyn, eds., A. Deepak Publishing, Hampton, Va., 1988, pp. 372–375.

or her necessary input data) starts an experiment, the task manager constructs a job control file invoking the necessary configuration of components and hands it over to the queuing system (www.openpbs.org). This system supervises the VL cluster's compute nodes and schedules job execution using a load-balancing strategy. It then supervises the job's further life cycle.

Integration and Administration Support

The VL platform offers support for integrating new components in the form of programs that

- Check interface descriptions
- Generate documentation skeletons from input descriptions
- Process finished documentation to extract metadata
- Update the component database using this metadata

Resource Monitoring and User Management

Via a protected Web user interface and auxiliary programs, the VL administrator role has access to the VL user database so it can add, remove, or modify user registrations. It can also supervise jobs running on the hardware cluster, supervise active user sessions, analyze log files, and administer the component database.

Dynamic Web User Interface Generation

Traditionally, turning a conventional program into a Web application was costly and tiresome because it involved lots of HTML and CGI coding. The VL platform automates this task for a large class of programs by generating Web user interfaces automatically from abstract descriptions formulated once per application by the "integrator" (the program's author or anyone having enough "behavioral knowledge" about the program). These descriptions capture only the essence of what data a user should request and when: the VL platform automatically adds presentational details. This also ensures that all VL Web user interfaces share a common style.

Command-Line Applications, Input Behavior, and Descriptions

Dynamic Web user interface genera-

Figure 3. The VirtualLab platform's architecture. When a program has been integrated as a component into the VirtualLab, it is accessible from a browser-only client via the application server.
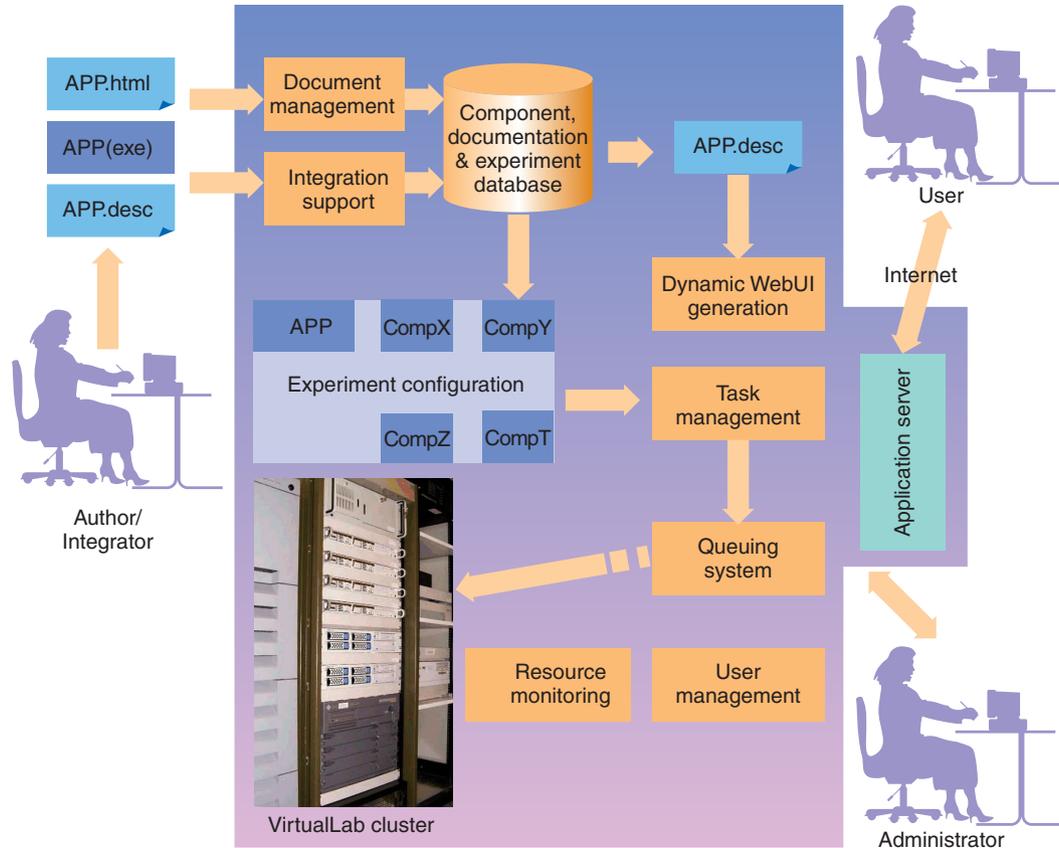
tion in the VL is tailored toward pseudo-interactive command-line applications. These applications operate in pure batch mode but can accept various structurally different sorts of input data sets. A common pattern is to read some control parameters and then, depending on their values, expect completely different input structures. This often occurs when a scientific program has evolved to cover multiple model variants, algorithmic approaches, or input data representations (such as coordinate systems). A good user interface should "know" the dependencies, request the discriminative input data elements from the user first, and then tailor the input request for the next interaction cycle accordingly. Our system offers a mechanism for abstractly describing all the relevant details of such applications' input behavior so that the VL can provide responsive dynamic user interfaces.

Input Data Element Descriptors
For each input data element, the integrator can specify the following information (see Figure 4):

- A *name* (for referring to that input data element later)
- A *precondition* describing when to request this input data element (a predicate over the values of input data elements already defined)
- The data element's *type*
- *Default values* to be offered in the Web user interface
- *Constraints* such as plausibility checks, allowed ranges, and so on (predicates involving this and, if appropriate, earlier input data element values)
- A *physical unit*
- A textual *annotation* providing some explanation about what is to be entered
- A *dimension*, if the input data element is a vector

Except name and type, which must be static, the remaining properties can symbolically depend on already known input values.

Limited space prohibits us from presenting all the description features in full detail here. Besides the input data element descriptors, there are *milestone* descriptors that let the system compute a value from already defined

input data elements (to factor out complex preconditions so they need not be repeated in multiple descriptors). There are also *block* descriptors that express how multiple input data elements are grouped into a sequence of Web pages (so the input data set is structured and overly long input form pages are avoided).

The Input Description Language

It became clear early in the project that a description language covering the needed abstractions would not be trivial. Name spaces, a type system, and symbolic expressions for specifying predicates were among the required features identified. Because these are standard elements of established, dynamic, high-level programming languages, we explored the idea of using an existing language in that role. Available language features proved perfectly suitable for representing the VL-specific abstractions, and this language reuse strategy showed considerable advantages:

- The effort for developing and maintaining a complex "homebrew" language (and translator) was saved.
- Features whose value we only later recognized were readily available in the base language.
- Integrators find a wealth of information resources for learning the language or when running into problems, and the knowledge gained here has independent value.

Python is our base language because it covers all the features needed, introduces little syntactic overhead, is easy to learn, and is already used in other parts of the VL platform.

A description file takes the form of a Python module containing a sequence of function calls; the input description

abstractions are available as preregistered classes, the constructors of which can be invoked like built-in functions. The description author needs no knowledge of these inner workings and must only be familiar with a small subset of Python. Nevertheless, because the standard Python interpreter (rather than some limited translator) processes the description modules, the base language's full power is available to expert authors. For example, people who know that descriptor parts needed in more than one instantiation can be factored out and bound to a name in advance, sets of similar descriptors can be generated in a loop, Python code blocks can be used to algorithmically express complex predicates, and so on.

How the Input Dialog Works

The description module, when executed, instantiates descriptor objects that are

collected in a data structure that is stored along with the component it describes. Each time a user supplies one or more input data values by submitting a Web form to the VL, this data structure is traversed, and all input data elements listed there receive a new internal state. Those that the submitted input data affect go to the "defined" state or to an "erroneous" state (on constraint violations). Because the new data might influence preconditions of subsequent input data elements, the latter could go from an "unrequired" to a "required" state (precondition fulfilled now, no user-supplied value available yet). The status values determine which input data elements will be re-

```
ID( Name      =        'Lambda',
    Precond   =        '_use_Lambda==1',
    Type      =        'FloatType',
    Default   =        '0.5',
    Constr    =        'Lambda > 0.0',
    Unit      =        'mu_m',
    Anno      =        'Input wavelength' )
```

Figure 4. Descriptor example. A single scalar input data element "Lambda" is specified with type, precondition, default value, constraint, physical unit, and annotation.
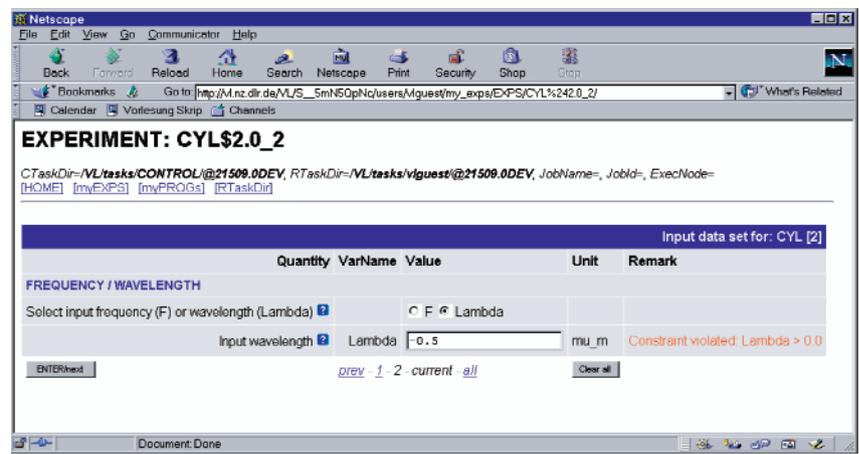

Figure 5. An example of a generated Web user interface. The input form field "Lambda" corresponds to Figure 4's example descriptor.
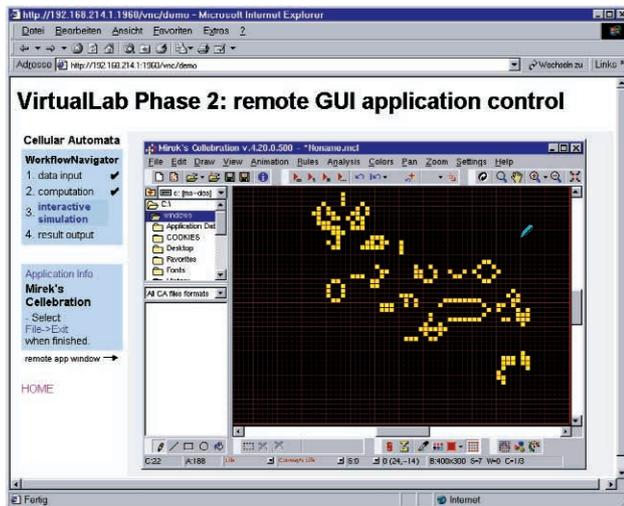
Figure 6. Using graphical user interface applications remotely. A Microsoft Windows-based cellular automata simulator is "remote controlled" from a browser window (using a VNC client Java applet).

quested from the user by the VL in the next screen, which of them will be rerequested with a warning to obey the constraints (see Figure 5), and which will no longer be requested because correct values are already stored. When no input data elements are "required" anymore, the input data set is complete, and the user can start the simulation run.

File-Level Input Description and Generation

Complementing the fine-granular description presented earlier, a coarse-granular mechanism specifies which input files a component expects. This information currently provides Web-based reviewing of input files just before execution and will organize multi-component workflows in the future. Specifying files to be HTTP-uploaded from a user is supported, too, for cases where form-based input would be inappropriate.

The input data set at the end of a form-based input dialog, which only exists in a Python object, still must be converted to the low-level data format that the component to be executed expects. The integrator must provide a procedure (in Python as well) that implements this conversion.

Output Handling

There is also a basic file-level component output description mechanism—a list of files the component in the task directory produces when successfully completing. The VL currently uses this information to prepare an archive file containing all results, linked from the generic results Web page shown after completion. The integrator can optionally have the component (or a script wrapping it) generate textual and graphical condensed result views; if present in the task directory, these are included in the result page as well.

Documentation Management

The VL defines a coarsely unified documentation structure that all components must obey. This structure has mandatory and optional elements, so an author is not required to supply everything at once. To avoid author lockout, putting severe restrictions on word processor preferences, for example, would not be an option. Thus we de-

signed a multiformat document management approach that works as follows: the standard structure is provided to the author in the form of a *skeleton* delivered in multiple text processor import-friendly formats—ASCII, LaTex, HTML, and RTF. An author imports the skeleton into a word processor, fills in the gaps, and exports the entire document as HTML (a standard text processor feature these days). The skeleton contains *pseudotags*—recognizable marker words in the exported HTML file that enable the VL to automatically extract the marked-up parts.

The document skeletons are dynamically generated even though the common document structure is fixed. Using information from the component description, the generated skeletons already contain prefabricated subsections providing basic textual descriptions of the input data elements and their Web user interface.

The common structure's static part includes component metadata such as author name, version number, and keywords as mandatory elements. When the documentation is integrated, these metadata are automatically extracted and directly used to feed the component database.

We plan to offer limited public access in the near future, bringing the aim of attracting new users closer to reality. Of course, resource considerations and the risk of denial-of-service type attacks are limiting factors.

For the immediate future, the main thrust of further development concentrates on support for workflows—recipes for executing components repeatedly or in coupled sequences. This feature will let users instruct the system to perform a series of experiments (for

example, using an entire range of parameter values automatically); it will also let them use components in pipeline-like combinations as well as in isolation, with the option to inspect intermediate results and to control further operation accordingly. We are also developing a Web-based assistant that can support the integration itself, which will use the workflow feature, too.

The VL's ability to construct Web-based user interfaces for command-line applications proved to be a successful concept, but scientific programs that already have a graphical user interface must be supported as well—for instance, data visualization packages. For enabling online graphical user interface application execution, we tested a thin-client approach based on the VNC package (www.uk.research.att.com/vnc). The original graphical user interface is used in a remote-controlled fashion here—an application's screen output is forwarded to the user via the Internet, and user actions (key presses, mouse movements) are transmitted back to the program. Our current work targets developing this approach further, integrating it with the workflow concept, and using it in connection with the WINE emulation layer (www.winehq.com) to support the execution of Windows applications (see Figure 6).

Likely fields of future development also include a common data repository shared by all users, personalized per component resource values, automated end-to-end testing, and syndication among multiple VLs.

References

1. S. Browne et al., "The Netlib Mathematical Software Repository," *D-Lib Magazine*, Sept. 1995; www.dlib.org/dlib/september95/netlib/09browne.html .

2. S. Browne, P. McMahan, and S. Wells, *Repository in a Box: Toolkit for Software and Resource Sharing*, tech. report UT-CS-99-424, Univ. of Tennessee, Dept. Computer Science, 1999.

3. C.A. Letondal, "A Web Interface Generator for Molecular Biology Programs in Unix," *Bioinformatics*, vol. 17, no. 1, Jan. 2001, pp. 73–82.

Thilo Ernst is the VL platform's lead architect. His interests focus on applying modern software concepts such as scripting languages and Web technologies in supporting applied research, scientific communication, and technology transfer. He received a diploma in mathematics from the Technical University of Dresden. Contact him at the German Aerospace Ctr., Remote Sensing Technology Inst., Oberpfaffenhofen, D-82234 Wessling, Germany; thilo.ernst@dlr.de.

Tom Rother is the head of the Atmospheric Processors section at DLR's Remote Sensing Technology Institute, where he also leads the VL project. He received his Dr.rer.nat. and Dr.habil. from the University of Greifswald. His research interests are in quantum statistics of charged particle systems and in electromagnetic wave theory. Contact him at the German Aerospace Ctr., Remote Sensing Technology Inst., Oberpfaffenhofen, D-82234 Wessling, Germany; tom.rother@dlr.de.

Franz Schreier is a staff scientist at DLR's Remote Sensing Technology Institute. His research interests include atmospheric radiative transfer modeling and inverse problems in atmospheric remote sensing. He received his diploma and Dr.rer.nat. from Ludwig-Maximilians-University Munich for work performed at the Werner-Heisenberg-Institute and the Max-Planck-Institute for Quantum Optics. Contact him at the German Aerospace Ctr., Remote Sensing Technology Inst., Oberpfaffenhofen, D-82234 Wessling, Germany; franz.schreier@dlr.de.

Jochen Wauer is a staff scientist at DLR's Remote Sensing Technology Institute. His research interests are in the kinetics of charged particles and in electromagnetic wave theory. He received a diploma in physics from the University of Greifswald and a Dr. rer. nat. from Humboldt University zu Berlin. Contact him at the German Aerospace Ctr., Remote Sensing Technology Inst., Oberpfaffenhofen, D-82234 Wessling, Germany; jochen.wauer@dlr.de.

Wolfgang Balzer is a staff scientist at DLR's Remote Sensing Technology Institute. His research interests include operational processing of atmospheric spectrometer data and the development of Web-based databases. He received a diploma in aeronautics and space engineering from the Federal Armed Forces University in Munich. Contact him at the German Aerospace Ctr., Remote Sensing Technology Inst., Oberpfaffenhofen, D-82234 Wessling, Germany; wolfgang.balzer@dlr.de.