# REPORT DOCUMENTATION PAGE

*Form Approved*
*OMB No. 0704-0188*



| 1. REPORT DATE *(DD-MM-YYYY)* | 2. REPORT TYPE | 3. DATES COVERED *(From - To)* |
|---|---|---|
| 26-09-2003 | Final Technical | 15-04-1998 - 15-02-2002 |

**4. TITLE AND SUBTITLE**

Intranet Delivery of Simulation-Centered Tutoring

**5a. CONTRACT NUMBER**

**5b. GRANT NUMBER**
N00014-98-1-0510

**5c. PROGRAM ELEMENT NUMBER**

**6. AUTHOR(S)**
Munro, Allen
Pizzini, Quentin A.
Johnson, Mark C.
Walker, Josh
Surmon, David

**5d. PROJECT NUMBER**

**5e. TASK NUMBER**

**5f. WORK UNIT NUMBER**

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
University of Southern California
Behavioral Technology Laboratories
250 N. Harbor Dr., Suite 309
Redondo Beach, CA 90277

**8. PERFORMING ORGANIZATION REPORT NUMBER**

**9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**
Office of Naval Research
800 N. Quincy Street
Arlington, VA 22217-5660

**10. SPONSOR/MONITOR'S ACRONYM(S)**
ONR

**11. SPONSOR/MONITOR'S REPORT NUMBER(S)**

**12. DISTRIBUTION/AVAILABILITY STATEMENT**
Approved for Public Release: Distribution is Unlimited.

**13. SUPPLEMENTARY NOTES**

20030930 121

**14. ABSTRACT**

The need for distance learning based on advanced simulation centered technical training, including artificially intelligent maintenance training using DIAG, motivated the development of iRides, an environment for delivering and authoring interactive graphical simulations and training. iRides makes it possible to transition training materials previously developed using the RIDES and VIVIDS research products for simulation centered training. Java programs that implement a simulation delivery system work in concert with authored graphics and behavioral specifications to provide interactive graphical simulations with training for distance learners. The iRides products were applied to a large-scale training development project on Basic Electricity and Electronics. This stress-testing of the research products revealed issues that serve to guide future research and development efforts.

**15. SUBJECT TERMS**
Simulation training, Distance learning, Advanced Distributed Learning, ADL, Authoring, Authored training, Authored simulation, iRides, VIVIDS, RIDES, DIAG

| 16. SECURITY CLASSIFICATION OF: | | | 17. LIMITATION OF ABSTRACT | 18. NUMBER OF PAGES | 19a. NAME OF RESPONSIBLE PERSON |
|---|---|---|---|---|---|
| a. REPORT | b. ABSTRACT | c. THIS PAGE | | | Donna Darling |
| U | U | U | | 26 | 19b. TELEPHONE NUMBER *(Include area code)* (310) 379-0844 |

# Intranet Delivery of Simulation-Centered Tutoring

## Final Report
## ONR Grant N00014-98-1-0510


Allen Munro
Quentin A. Pizzini
Mark C. Johnson
Josh Walker
David Surmon

Behavioral Technology Laboratories
University of Southern California
250 No. Harbor Drive, Suite 309
Redondo Beach, CA 90277

(310) 379-0844
munro@usc.edu
http://btl.usc.edu/

# Final Report: Intranet Delivery of Simulation-Centered Tutoring[1]

Allen Munro
Quentin A. Pizzini
Mark C. Johnson
Josh Walker
David Surmon

University of Southern California

## Summary

The first goal of this project was to develop a capability to deliver interactive technical training, including artificially intelligent maintenance tutors authored using DIAG and the RIDES/VIVIDS authoring tools. The research products RIDES (Munro, Johnson, Surmon & Wogulis, 1993; Munro, 1994; Munro, Johnson, Pizzini, Surmon, Towne & Wogulis, 1997) and VIVIDS (Munro & Pizzini, 1998) are integrated authoring and delivery software systems for creating and presenting interactive simulation-centered training environments. These systems were used to create DIAG (Towne, 1996), a tool for providing intelligent instructional guidance during simulation-based maintenance troubleshooting training.

Two major technical advances were required to facilitate the eventual transition of the RIDES, VIVIDS, and DIAG research products to meet Navy training requirements. First, it was essential that the software should be able to run on Windows systems, and ideally other operating systems, as well.[2] It was necessary that the new system be platform-independent, so that it could be delivered on a variety of platforms found in Navy environments, including all modern Windows operating systems, as well as Unix and Linux. Second, the increasing importance of intranets and of the Internet for military training called for the capability of on-demand distributed learning. A network-deliverable implementation of a simulation-centered learning platform was called for. If this distance learning software could be made compatible with the RIDES/VIVIDS system, then previously developed training applications, such as DIAG, could be made available as distance learning modules.

---

[1] This is the Final Report on research conducted under Office of Naval Research Contract No. N00014-98-1-0510, *Intranet delivery of authored simulation-centered tutors.*

[2] The original RIDES project (supported by Air Force Contract No. F33615-90-C-0001, which included Office of Naval Research co-sponsorship) had requirements that could be provided only by Unix operating systems when that effort began in 1990. Hence, RIDES and its immediate successor, VIVIDS were developed using Unix and a Unix-only application framework, Interviews (Linton & Vlissides, 1989; Vlissides, 1990).

A new simulation training delivery system, to be called *iRides*—for *Intranet RIDES*, was a target research product for this project. iRides was to have the following two major characteristics:

- Network-safe intranet access to simulations for training
- Ability to deliver simulations authored with RIDES/VIVIDS
  - including the ability to run DIAG, the most complex application of RIDES

Development of iRides required a completely new implementation of a RIDES/VIVIDS style simulation system, because the C++ code of VIVIDS could not be directly adapted to support intranet delivery. It would be necessary to use the Java programming language to create the intranet-deliverable iRides simulation delivery software. (The feasibility of this approach had been demonstrated in a prototype simulation player, called jRides, which was developed in Java in 1997. jRides implemented the expression syntax of VIVIDS, and a subset of its simulation language functions, using a rudimentary approach to simulation graphics.) Although the transition to a complete implementation in a new programming language imposed a significant design and implementation workload, this necessity also presented an opportunity to advance the state of the art by introducing additional capabilities into the new simulation delivery system. These capabilities are described below in the *Results* section. In brief, however, these enhancements included

- Additional data types for simulation attributes
- An improved, extensible simulation language with flexible expression syntax
- Support for cloned simulation objects, in order to provide simulations that have on-the-fly creation of new objects
- Improved graphics capabilities
- Simulator support for interactive debugging in iRides

In order to run original VIVIDS simulations, iRides had to be able to read files produced by VIVIDS. The original VIVIDS file format was a complex binary format based on a commercial library for representing complex reference relationships. There were several problems with continuing to use this file format for the new iRides research product. One of these was that binary file formats effectively render all authored information vendor-dependent. Future computing systems would not be able to even display the content of such files if those systems did not support the authoring tools that created them. There has been a long history in the aviation community of large-scale training products becoming obsolete, not because the content of those trainers became outmoded, but because it became impossible to buy computers that could run the antique operating system and software of those systems. Use of a human-readable text format would improve the chances of costly content development remaining usable for the foreseeable future. VIVIDS itself was modified so that it would be able to output previously authored simulations in the iRides text file format.

The first goal of the project, implementing a multi-platform, network-delivered simulation environment for training that could deliver applications as complex as DIAG, was achieved very well during the course of this project.

Authoring iRides simulations in the native iRides context, rather than only in VIVIDS was a second major goal of this project. Native authoring would make it possible to create new distance learning simulations without having to make use of VIVIDS and the Unix platforms on which VIVIDS runs. New editor interfaces were developed in Java for the iRides environment, including object lists, object data editors, attribute data editors (including constraint editing), and event editors. This simulation behavior authoring system was in a very usable state at the completion of this project. It has since continued to be improved under Future Naval Capability (FNC) ONR funding, as is briefly described at the end of this report.

A third major goal of this project was providing support for the transition of BEESIM basic electricity and electronics lessons developed with classic VIVIDS to the iRides environment, and, later in the project, to contribute to the BEESIM development effort. This effort revealed a number of problems, including the need to anticipate expectations in the computer based instruction community that all authoring systems must make use of a page-turning model of presentation, the need to make notions of 'lesson' and 'course' more flexible, and the difficulties of implementing an entirely new training delivery framework, based only in part on iRides, in a very short period of time. Certain of the lessons of this effort have resulted in improvements to later versions of iRides, which are still under development at the University of Southern California. These impacts are briefly described in the final section of this report, *Continued iRides Development.*

## Background

Modern computer-based learning environments can benefit from *action centered training* techniques. In action centered training, a tutorial entity observes student performance in an action context and carries out pedagogical activities in that context. *Action contexts* are often simulations, but other action contexts are possible. In embedded training, for example, complex equipment systems based on computers can serve as action centered learning contexts. Interactive simulations and embedded training systems can help to ensure that job-related performance skills—as opposed to mere test-taking skills—are acquired as a result of training.

Many earlier research projects on advanced tutoring systems have incorporated simulations that were developed using low level tools (i.e., programming languages) to develop both the tutorials and the simulations. Reliance on such low-level development techniques naturally makes simulation-centered tutoring extremely expensive. It can also make it very difficult to determine what features of a particular tutor are responsible for its efficacy (or lack thereof). Authoring systems, by ensuring a uniform quality of low-level instructional interaction and by providing easily edited and modifiable tutorials and simulations, can make it possible for developers to experiment with different high-level approaches to training in a given domain.

Consider an environment that is designed to help aircraft mechanics learn how to carry out their jobs. One module might help students learn about an aircraft's AC Power

System. In addition to presenting textual, audio, video, and static pictorial resources that present the appearance, functions, and use of the AC Power System and its components, the learning support environment could also make use of an action context such as the interactive graphical simulation depicted in the figures below. This simulation of a cockpit control panel can be a behaviorally realistic environment for demonstration and practice.

There are two separate issues to be aware of in this example. First, what should be the characteristics of the simulation? Second, how should the training component of the learning environment be able to interact with the simulation to facilitate learning?

*An interactive live simulation—an action context for learning.*

A *live simulation* is one that permits arbitrary sequences of actions by the user. In moderately complex simulations, there may be billions of possible action sequences that users can follow in using the simulation. Each action that a user takes causes a set of effects analogous to those that would be observed in a real device. Consider the aircraft AC Electric Power control panel shown at the right. This panel is used to control AC power systems on certain naval aircraft.

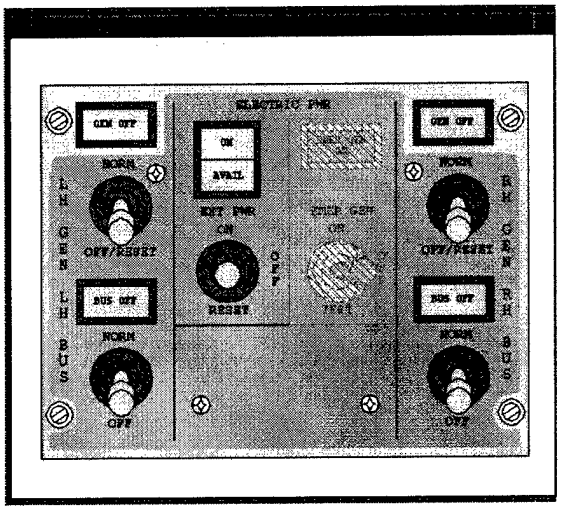


*Figure 1. AC Power Simulation, Initial State*

If the user clicks on the lower half of the External Power toggle switch, that switch will be put into the Reset position. This switch position activates a lamps test circuit that causes all the lights on the panel to be yellow.

To actually use the AC power system, it is necessary to put the External Power switch in the On position, and to put both the left and right generators on, and both the left and right hand bus switches on.

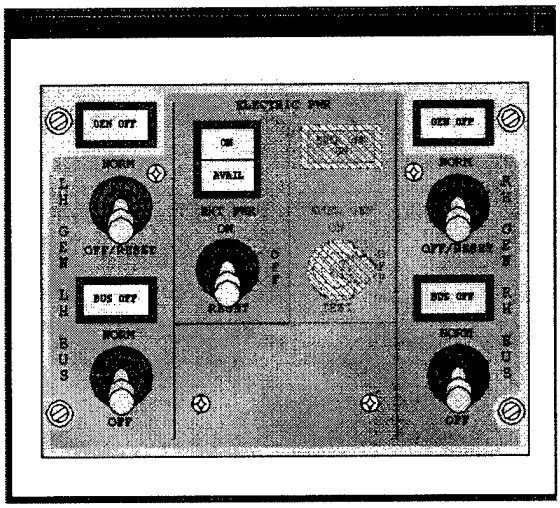


*Figure 2. AC Power Simulation*

In a live simulation, the results of any sequence of actions will be correctly displayed. In the picture at the right, the user has turned the External Power switch on and has engaged the left hand generator and bus, but has not turned on the right hand generator and bus. No matter what sequence of operations is carried out, this simulation shows the consequences at each stage of the process.

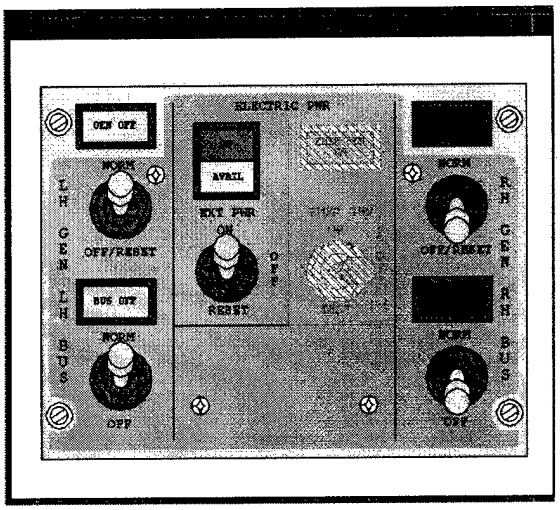

*Figure 3. AC Power Simulation*

**Action-centered training.** In action-centered training, a training system can monitor student actions, can provide instructional feedback in the action context, and can carry out actions itself. The three figures below illustrate several types of low-level interaction that a coach can have in an action environment. These interactions include monitoring student actions, controlling simulations to undo student actions, giving students informative feedback in the context of the simulations, demonstrating actions in the simulation, and monitoring states of the simulation.

**Monitoring actions.** An action-centered trainer must be able to observe a student's manipulations in the action context. It must be able to evaluate actions. When a correct action is taken, it may be appropriate for the training controller to give the student informative feedback or encouragement. When an incorrect action is carried out, it may be useful for the tutor to inform the student that the action was not correct.

**Control—Undoing student actions.** In addition to providing feedback about incorrect actions, it is sometimes necessary for a tutor to negate the effect of a student action during training. If actions are not 'undone', it may be difficult or impossible to step the student through the proper sequence of actions that is to be learned.

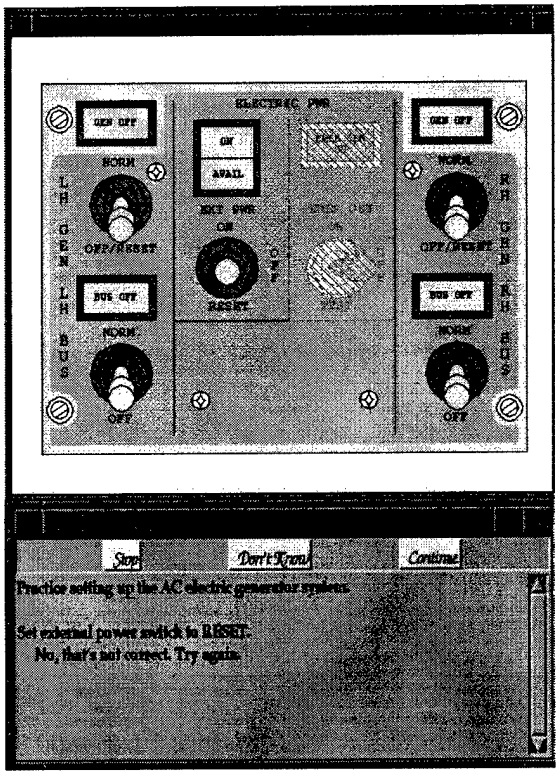


*Figure 4. Monitoring Actions and Controlling a Simulation for Training Purposes*

**Feedback in the action context.** In order to draw a student's attention to particular objects or elements in an action context such as a simulation, it may be helpful to graphically highlight an object. In the figure at the right, the left hand generator switch is made prominent by flashing rapidly between two distinctive colors. The simulation view carries out this highlighting process when directed to do so by a training controller object.

**Demonstrating actions.** A tutor sometimes needs to carry out the steps of a procedure in order to teach a student how to perform the procedure. In Figure 5, the student is first prompted to indicate that he understands what is being referred to. Then the tutor directs the action context to emulate the desired action.

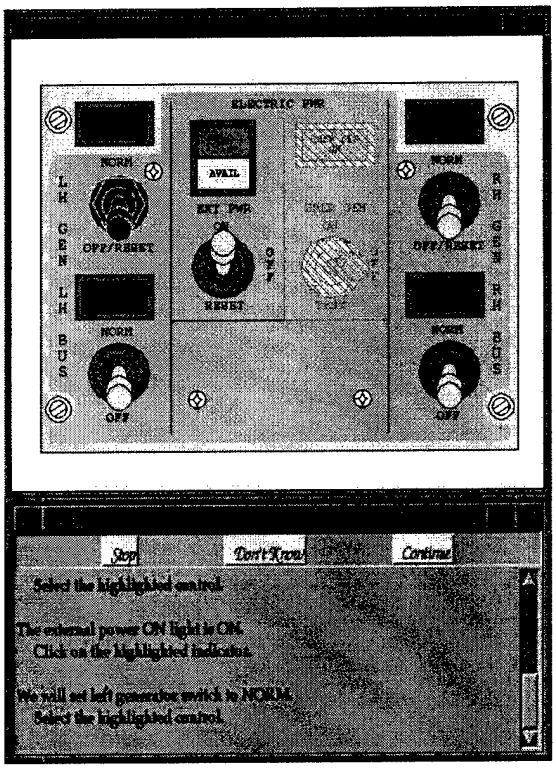


*Figure 5. Instructional Feedback in a Simulation Context*

**Monitoring states.** In addition to monitoring discrete student actions, a trainer must also be able to monitor a simulation or other action context for defined states. That is, when a particular state occurs, it may be appropriate for the trainer to detect that state occurrence and to respond instructionally. In the figure at the right, the trainer first directed the student to 'set up the right generator and bus'. It then allowed the student to carry out any actions the student chose. When the target state was achieved, the trainer detected the state and gave the student feedback to indicate that the desired state had been achieved.

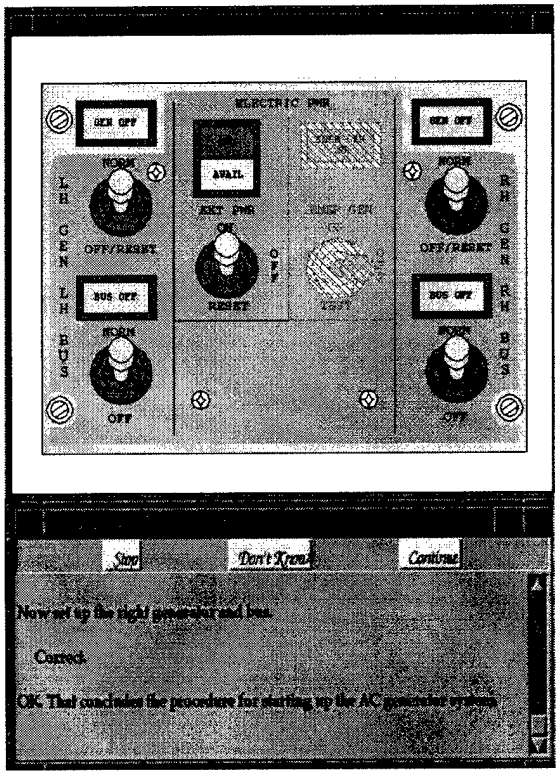


*Figure 6. Monitoring Simulation States*

## Components of an Action Centered Learning Environment

As a result of many years of experience in action-centered learning contexts (Munro, 1994; Munro, Johnson, Pizzini, Surmon, Towne, & Wogulis, 1997; Stiles, McCarthy, Munro, Pizzini, Johnson, & Rickel, 1996) we have developed a consistent organizational system for viewing all such environments. From the student user's viewpoint, there may be four major visible types of components in an action centered learning system.

- **The visible action context.** In the case of simulations, this type of component is one or more views of the simulation. In the figures above, the views of the cockpit panel are this type of component.
- **Presentation interface.** The window with instructional text (In Figure 6, the last line says "OK. That concludes the procedure for starting up the AC generator system.") is an example of a presentation interface. Other examples of presentation objects include video players, audio presenters, and text-to-speech presenters.
- **Commands.** The buttons labeled *Stop*, *Don't Know*, and *Continue* are examples of commands. These are interface objects that make it possible for a student to carry out meta-interactions, such as asking for help or quitting a session.
- **Entries.** A coach sometimes must ask a question of a student that cannot be answered by carrying out an action in the action context (such as the simulation view). Interfaces for gathering answers to questions are called Entries. The menu at the right contains a set of choices to be offered to a student when the question is posed, "What is the value of the External Power Available light" in a simulation context such as that shown in Figure 2.


**The invisible components of an action centered learning environment.** In addition to the visible components, an action centered learning environment also has important invisible components. These include:

- **The Controller.** The Controller component is the tutorial object that controls the course of the learning experience. A Controller makes use of the other components, deciding whether to respond to commands, directing presentations, and monitoring and affecting the simulation (or other action context).
- **The Behavior Model.** The component of the action context that determines how the simulated system works is called the Behavior Model.
- **The Student Data Framework.** Data about the student is maintained for the use of the Controller. The Controller can direct that information be stored and can retrieve information about the student for use in deciding how to proceed with the learning experience. The iRides system includes a mechanism for storing arbitrary data during training. This means that it does not at present include a specific type of active student model, although the architecture could support the inclusion of such a component.

While not every action centered learning system has all of these components, this broad design may provide a useful perspective for analyzing or designing the structure of such systems.

## Instruction in the Context of Behavioral Models

One of the key concepts in VIVIDS is that instruction takes place in the context of an authored graphical model of a particular (usually complex) man-machine system that represents all or many of the relevant characteristics that are to be learned. For example, if the domain is the operation of a certain item of medical equipment, then the appearance, behavior, and proper usage of that equipment is the domain to be learned. Figure 7 shows a student environment with two windows open. The lower window is the student instruction window, where instructional text and remedial directions are displayed. The upper window is one of several windows in this simulation that contain depictions of the domain of interest—here, a pulse oximeter. Simulation scenes, such as the one with the title "Pulse Oximeter" in the figure, provide graphical models that are the context for all instruction in VIVIDS. Many elements of instruction can be automatically generated by VIVIDS because they are developed in the context of a structured domain model.

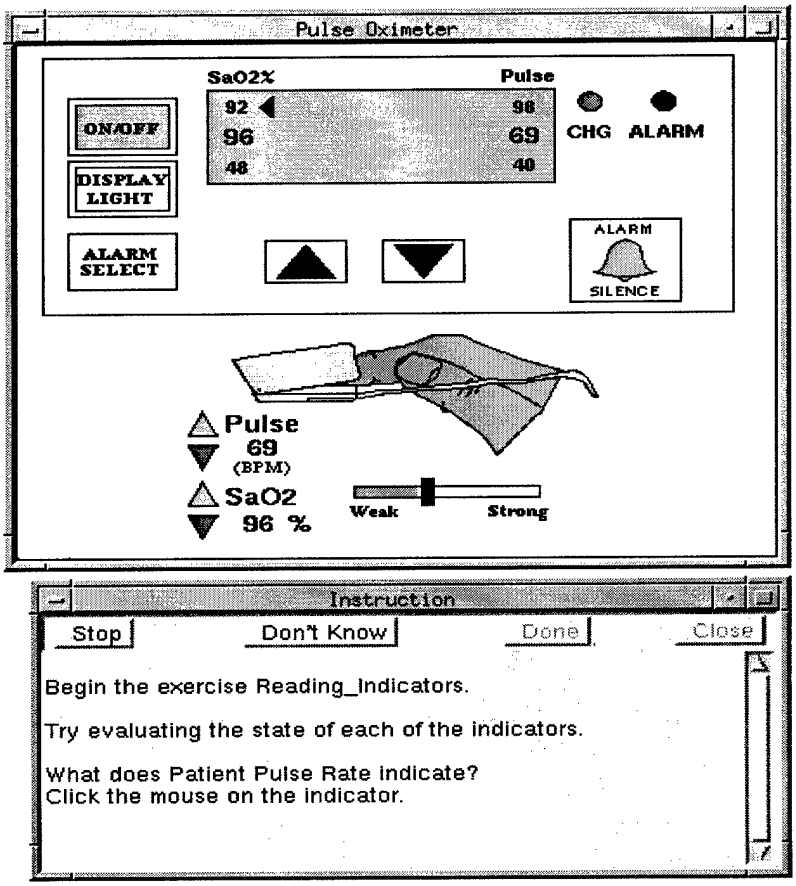


*Figure 7. A Student View of Instruction in the Context of a Graphical Model*

Authors can build complex graphical models by pasting simulation objects into their scenes. They can also draw objects directly onto the scenes, using a palette of drawing tools, and can specify rules that control the values of object attributes. The finger in the figure above is one such object. Authors can open an *object data view* of the object, such as the one shown in the figure below. An object data view of an object shows its name, together with a list of its attributes with their values and their rules, if they have any. In this figure, there is a rule for the attribute named *pulse*. The present value of this attribute is determined by the rule, which refers to the attribute of another object.

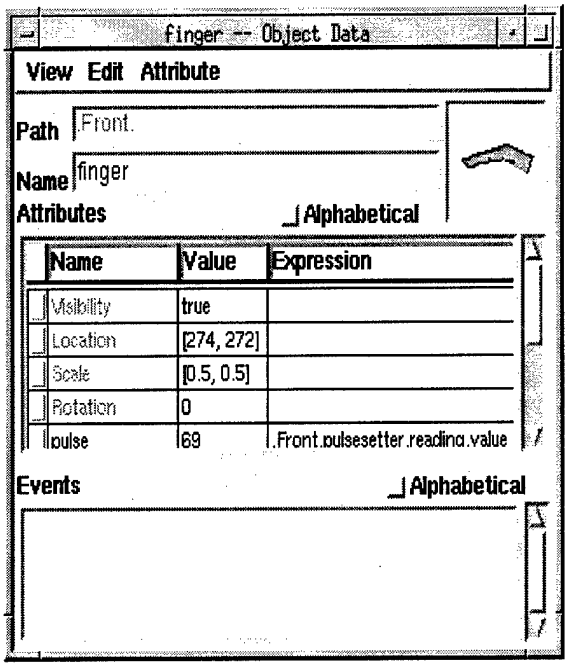


*Figure 8. Object Data View for a Graphical Object*

Some attributes directly control the appearance of graphical objects. When the values of such attributes are changed, objects may disappear, move, stretch, rotate, or undergo other visible changes.

## The iRides Simulator

Constraint-based programming languages provide a number of benefits over conventional declarative languages. Foremost among these is the elimination of much of the burden on the programmer to determine the flow of control of program execution. Constraints are pieces of code that are executed when values they refer to change. In conventional programming languages, the programmer is responsible for determining the order of execution of program statements. In a pure constraint language, there is no "order" per se: The system of constraints represents a steady-state model of the system being simulated. When that system is perturbed by outside influences such as mouse or keyboard input, the constraint-firing mechanism in the execution environment is responsible for determining when each constraint must be evaluated and in what order. Constraint-based languages have a number of advantages, including a reduction in the complexity of the programming task, because flow of control is not ordinarily the responsibility of the

constraint author. A constraint-firing mechanism in the constraint execution environment is responsible for determining when each constraint must be evaluated (executed). Several constraint-based programming languages have been developed (Leler, 1988; Munro, Johnson, Surmon, & Wogulis, 1993; Munro, 1994). Examples of constraint-based interactive graphical environments, include Sketchpad (Sutherland, 1963), Thinglab (Borning, 1979), and a spreadsheet-based graphics system by Wilde and Lewis (1990).

Constraint-based programming languages, like other programming languages, can be extended through the definition of new functions and procedures. For example, if a programming language does not have a square root function, a programmer could write a new `sqrt()` function in the constraint language that returns the square root of a numeric parameter.

#### Interactive Representations Controlled by Authored Constraints

Some systems for creating interactive graphical environments provide a constraint-centered approach to authoring behavior. In such systems, rather than specify what should take place when crucial events happen, an author specifies what relationships are to endure among values in a system. An example of a constraint-based environment with one-way propagation of effects is a spreadsheet authoring application, such as Excel. An author specifies that the value shown in cell D is the sum of the values in cells A, B, and C. When any of these values is changed by a user, the value of D is also changed. The author does not have to specify that the event of a value change in A or B or C should invoke the computation of D.

In VIVIDS and iRides, constraints are written as expressions. Whenever any value that is referred to in a constraint expression changes, the expression is evaluated and a new value for the attribute is determined. It is not necessary to explicitly state that an expression must be re-evaluated whenever the a value referred to in the expression changes. Authors need not concern themselves with when a value needs to be recomputed if that value is determined by a constraint.

## Results and Discussion

### Network-safe access to simulations and training

The selection of the Java programming language, together with the utilization of both applet and Webstart technologies, helped ensure the implementation of network-safe access to simulations and training in iRides. Suppose that iRides had been implemented as an independent program—one that was not constrained to operate within the Java environment but one that instead had full access to the operating system interfaces on the students' computers. When that program was downloaded along with a simulation specification, the authored simulation system would have access to some powerful and potentially dangerous capabilities on student machines. An inimical or incompetent simulation author could conceivably write a simulation that could damage files on the

student's computer. Because Java forbids unsafe access in the context of ordinary applets and Webstart applications, student computers are protected from these problems. Of course, there may be cases in which it is necessary for a training application to modify local files. There are two primary ways that this can be accomplished using iRides. One is to make the iRides training object a Trusted Applet or a Trusted Webstart application. This can be managed by the system administrator at the student's site. The second way is to use an application version of iRides, which has full application prerogatives on student stations. This approach is presumably acceptable at school sites, where technical support personnel and training vendors can test and vouch for the beneficence of the training simulations that will be installed.

## Ability to Convert Simulations Authored in RIDES and VIVIDS for iRides

Numerous RIDES and VIVIDS simulations have been exported to iRides file format and tested in the iRides application. These include the B2 Landing Gear simulation, the Shipboard High Pressure Air Compressor simulation, and the simulation of the Gas Turbine Engine Control System on Arleigh Burke class destroyers, which was developed under an earlier Virtual Environments for Training project. In most cases, no changes were required to obtain identical behavior and virtually identical visuals in the iRides versions of these simulations.

### Conversion of DIAG training to iRides

The most thorough possible test of the iRides simulation delivery system was the ability to deliver the DIAG intelligent tutoring system. Almost every capability of the RIDES/VIVIDS simulation system is utilized by DIAG and must be supported in the iRides delivery environment. However, in order to support DIAG it was not necessary to re-develop all the RIDES/VIVIDS instructional system, because DIAG makes use of simulation graphics and the simulation behavior language to deliver instruction.[3] [4]

Early testing of DIAG with iRides revealed that DIAG made use of one aspect of VIVIDS that is neither part of the simulation system nor part of the lesson control system. That aspect is the VIVIDS *Knowledge Unit* mechanism. (Knowledge units are networked text bodies with indexing attributes and links that attach them to elements of the simulation.) Features originally not planned for iRides were implemented in order to support those characteristics of DIAG that made use of VIVIDS knowledge units.

### Conversion of authored BEESIM lessons to iRides lesson structures

Initial content development for the BEESIM course was done by third-party instructional developers using VIVIDS. However, the intent was for those lessons to be deliverable over networks on a variety of computers. It was therefore necessary to develop a way to

---

[3] During the period of the work reported here, a parallel project supported primarily under an extension of U. S. Air Force contract F33615-90-C-0001 funded the development of an instructional delivery system that works with the iRides simulation delivery software.

[4] A second implementation of DIAG has since been developed using Toolbook. See Towne (1998, 1999).

convert those VIVIDS lessons into equivalent iRides lessons. This was done by adding to the VIVIDS code the ability to output the VIVIDS lesson as a file that in the format that iRides uses for lesson specifications. In addition to being able to execute a Save command to save the VIVIDS simulation and lessons as a single (binary) classic VIVIDS file, the author can now also save the simulation as an iRides structured text file and save the lessons as an iRides lesson file, that is, as an LML file.[5] So, in order to run the previously developed BEESIM course in iRides, a VIVIDS author needs to

- save the simulation in iRides format (a .jr file)
- save the lessons in iRides format (.lml files)
- copy these files into the appropriate directory or directories
- make the standard java call to start running a lesson

The conversion writes out to file a standard header section for an LML file. Then, for each part of the VIVIDS lesson (e.g Freeplay, Present Text, Find Object), it writes out a chunk of LML code that would behave the same way as the VIVIDS segment. Initially this produced very large LML files, which were difficult to read and debug. This observation motivated additional enhancements in later work described in the section *Continued iRides Development.*

## Enhancements to the iRides Simulator

### New data types

In VIVIDS, there were six types of attribute values that were available to the author: number, logical, text, color, point, and pattern. It became apparent some time ago that some types of simulations were very difficult to build when restricted to these value types. Consequently, under this contract, two new data types were implemented: arrays and object references.

Introducing arrays actually simplified the data types of iRides. It was no longer necessary to have separate Point and Color value types for example, because they can be treated in iRides as two-element and three-element arrays of numbers. By avoiding special data types like 2_D_Point (for representing locations on two-dimensional surfaces) and 3_D_Point (for expressing locations in three-space), simulation code can be made more general. Instead of using a specific 2_D_Point data type, a simulation could be built using a two-cell array, and instead of a specific '3_D_Point or 'color' data type, a three-cell array could be used. Further, there were some simulations written using VIVIDS that really needed arrays of other sizes; the implementation of these simulations without arrays was extremely convoluted. Generalizing further, the individual cells of an array can be filled with any of the data types, not just numbers, including arrays. The only restriction is that all cells of an array must be of the same type—all numeric, or all text, etc.

---

[5] Saving lessons in the LML file format was funded primarily by the Air Force contract F33615-90-C-0001.

It was often found in writing a simulation that the easiest, most efficient, and least error prone way to write it would have been to have references to objects as a data type. An example would be if you wanted to perform some operation on several numeric attributes of an object. It was possible to do this in VIVIDS, but it was painful for the author. Moreover, if the name of the object was later changed, none of that code would work any more. But if you could set some attribute to have the value of the reference to an object, then writing that code would be greatly simplified, and if the name of the object was subsequently changed, the related rules would not be affected. iRides now implements attributes so that their values can be an object reference; also the value of an attribute could be an array of object references.

#### Extensible simulation language

In VIVIDS it was impossible for the author to add a new function or procedure to the programming language. Take for example the function 'cubeRoot'. VIVIDS did not implement this function. An author could implement this within one VIVIDS simulation, but it would not be available in other simulations. If 'cubeRoot' was needed in several simulations, it would need to be written within each one. The only way to make that function universally accessible would be to modify the source code of VIVIDS and recompile it, an onerous task and one that would only be possible if the user had access to the source code. Moreover, since VIVIDS uses an interpreted constraint language, such a user-defined function would typically run much more slowly than the core 'native' set of functions and procedures of the language. This is true because the body of a user-defined function will itself be interpreted, while the native functions of the language will be executed from a compiled representation. The structure of the iRides simulation language now permits programmers to extend the set of 'native' functions of the iRides constraint-based language. So now a programmer could write a new cubeRoot() function that returns the cube root of a numeric parameter. This makes it possible to develop faster, more efficient routines. These new user functions become part of the language. In fact, the standard iRides functions are defined in exactly the same way, and they are invoked in precisely the same way as the core set of functions. Unlike many interpreted simulation systems, there is no execution penalty to this flexibility, since the user functions are compiled code, not interpreted code. iRides has several techniques that together support authored simulations with the 'native' extensibility of constraint-based programming languages in the context of program execution environments, such as Java, that support reflection. These innovations are:

1. Controlling representational objects with an interpreted constraint language,
2. A method for adding 'native functions' to such an interpreted constraint language,
3. A technique for specifying the number and data types of the parameters of a user-defined constraint language's functions and procedures, and
4. A method for specifying the external triggers of user-defined constraint language functions and procedures.

When a user wishes to extend the constraint programming language with a new function, he or she must create a new class derived from the constraint language function base class. This derived class must specify the behavior of the new function by overriding the compute() method.

### Object templates and clones

In some VIVIDS simulations it was desirable to create a set of similarly behaving objects, for example, each of the objects would have its own location, but the movement of all the objects might be governed by the same rule. This could be done by creating one object then making copies of it and modifying the location of each of the copies. A problem occurred if the author later decided that the rule governing movement needed to be modified; it then became necessary individually to edit the movement rule in each of objects. It became clear that a notion of object templates and clones would be useful; this was implemented for this contract. The process the author can now use is to create one functioning object, complete with behavior rules, and then define a template based on that object. Then clone objects can be created using that template as a basis. Each clone inherits the rules specified in the template. Then if a rule in the template is later modified, the corresponding rule in each of the clones is automatically modified in the same way. If an attribute of a clone does not have a rule specified by the template, a rule can be written for that clone, and the other clones will not be modified in a parallel fashion. Also, if a clone attribute does not have a rule attached to it, its value can be changed at will. Using these mechanisms a set of similarly behaving objects can easily be created and modified, but the clones can be individualized to some extent by means of their own private rules and values.

### Graphical improvements

Several major new graphic features were developed for iRides under this contract, features that were not present in VIVIDS.

- Full support for alpha-channel transparency, including animating transparency (*i.e.*, fading in and out).
- Gradient support, both cyclic and non-repeating.
- Support for full-color JPEG format images, as well as the older GIF format.
- Support for textures (using an image to fill an object, instead of a solid color) using the above image types, for all shapes.
- TrueType font support.
- Full antialiasing, for both shapes and text. For performance purposes, antialiasing can be turned on and off for shapes or text independently.
- Scenes are now scaleable, so zooming in on an object or area is now feasible.

Textures, fonts and transparency in particular are the three most important advances. With these three new features, many new graphical effects are available that were either very difficult or simply impossible before can now be employed in iRides simulations.

### Simulator Support for Interactive Debugging

VIVIDS included a runtime interactive debugger. A stepping debugger can be a very useful authoring aid, because it allows an author to trace the execution of the constraints and events of a simulation in a step-by-step manner. The author can also test for the values of attributes and expressions during the execution sequence. These capabilities

have been found to promote the rapid development of accurate and predictable behaviors. During this project, 'hooks' were introduced into the simulator to support debugger notification and the process of stepping through a simulation's rules under the control of the author. A debugging interface was designed that would introduce additional features to improve the usefulness of the simulation debugger in iRides. In particular, the new debugger would support three new features not available in VIVIDS.

- Stepping into author-defined event functions. In many cases, authors do not need to see the step-by-step execution of the statements of a well-defined event body that has already been thoroughly tested. Introduction of a *Step Into* command makes it possible for authors to step over their reusable behavior components specified in events under ordinary circumstances.
- Stepping out of event functions. Sometimes an author wants to step into an event while tracing a simulation in order to observe a portion of its execution. Once the statements of interest have been executed, the user can choose *Step Out* to complete the remaining statements of the event without stepping through them all one-by-one.
- The evaluation of arbitrary expressions. In classic VIVIDS, an author would have to create dummy attributes and give them constraint rules in order to evaluate arbitrary expressions. A special expression evaluation interface was designed for entering and evaluating arbitrary expressions during the debugging process.

This design and the underlying implemented features were rapidly exploited in follow-on work on the iRides system. See *Continued iRides Development*, below.

## Editors for Authoring Simulation Behavior in iRides

A view of the major editors of the iRides authoring application is shown in Figure 9. This Java application can run in all the modern Windows environments (Windows 98 and later). In the application version, authors can have access to the internal behavior data of a simulation, which they can modify on the fly. In Figure 9, the leftmost window is the top data view of the behaving objects of an iRides simulation. This is a hierarchical list of objects and the scenes that contain them.

Each scene contains a group of behaving objects, and those objects themselves may be groups, containing other objects. An author can open objects in the list to view their component objects. Any object can be selected at any level, and an object data view can be opened. The lowest window in Figure 9 is an example of an object data view. This object is named Readout and is part of the CurrentReadout object on the Visualization scene. Every object has a number of attributes. The rightmost window is a data view for an attribute named Current that is part of the same Readout object. The attribute has a rule that determines its value. This rule can be modified in the data attribute view. As soon as the Accept button is pressed, the rule is parsed and formatted to reflect the parser's understanding of the behavior specification. At the same time, the simulation is updated with the new rule, and the interactive behavior of the simulation will immediately reflect the new or revised rule.

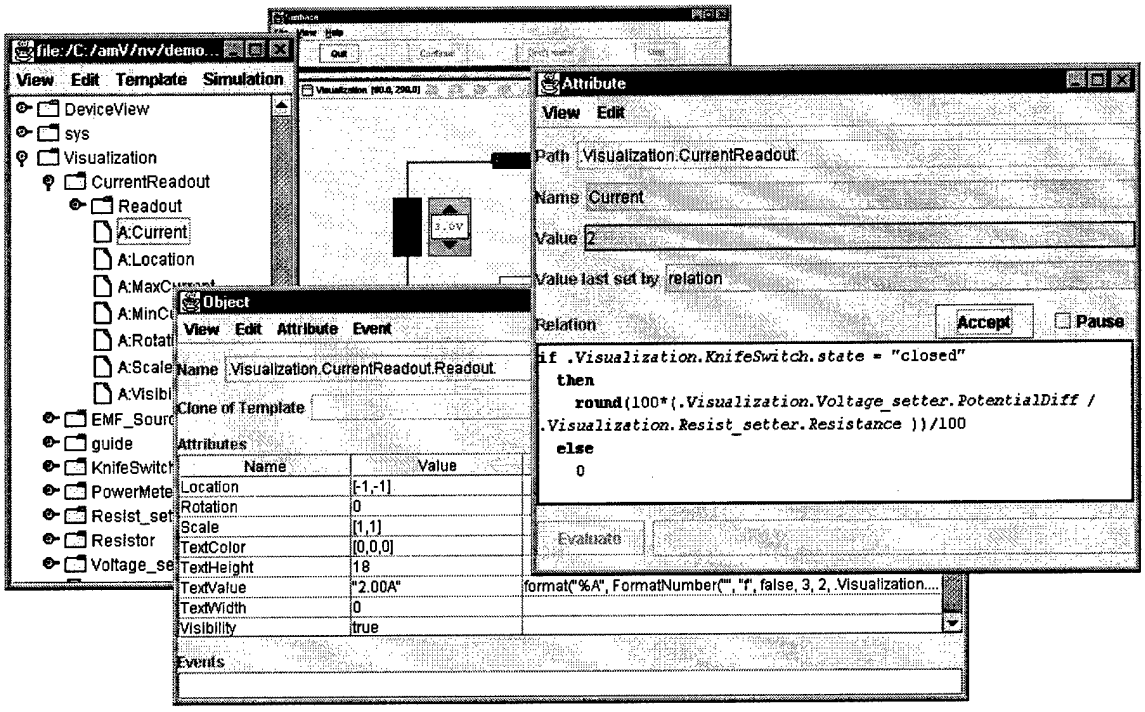


*Figure 9. The Behavior Authoring Interface of iRides*

Authors can us the File menu's Save and Save as… commands to save the entire simulation specification file, retaining any changes made using these editors.

## Support for BEESIM

### Phase 1: Serving as a Resource to the BEESIM Designers and Authors

After more than a year of development by a CBT vendor, the first author was given the opportunity to serve as a resource to the design and development staff of that company who were working on the BEESIM project.

Several challenges to the success of the BEESIM goal became evident, including these:

- The third-party developers of BEESIM made unexpected uses of VIVIDS features in order to create page-turning user interfaces, rather than the more normal (for VIVIDS and iRides) teaching-in-the-context-of-a-simulation interface. They also developed some specialized assessment interfaces that were not at all the originally intended type of application for VIVIDS. These authoring practices made the transition to iRides more difficult.
- Some computer programmers on the third-party BEESIM development team apparently did not understand the ability of the simulator to automatically maintain constraint relationships among attribute values. They were drawn to recreate features of the simulation engine using the VIVIDS simulation language. Confusion and inefficiencies resulted.
- The VIVIDS structure of 'courses' and 'lessons' did not match well with the uses to which simulation was put in BEESIM. In particular, the detailed design

of BEESIM sometimes called for the use of a succession of simulations in a single lesson. Classic VIVIDS could support only one simulation for a lesson. As a result, many small lessons had to be stitched together with the VIVIDS 'course' mechanism in order to create a single lesson of thirty minutes or an hour. This result stimulated the later development in iRides of the capability to close a simulation and open new ones within a lesson. In addition, a scalable approach to pedagogical structure was added to iRides, as is described in the next section of this report.

- Some BEESIM lessons contained multiple copies of complex behaving objects on different 'pages' (different parts of the 2D simulation world). More experienced VIVIDS developers might have moved and reconfigured the objects for use at different points of the lesson, using standard VIVIDS features. As a result of the explosion in behavior rules (a complete set for each such object), VIVIDS lessons in BEESIM sometimes bogged down, and crashes often resulted due to data reference errors. This was another inducement to the original BEESIM developers to divide lessons up into very small, more reliable chunks. (iRides, which makes use of the automatic recovery of unused memory in Java, does not suffer from this fragility.)

These findings led to a number of redesign decisions in iRides. For example, in order to support larger lesson structures, iRides was redesigned to support closing and opening a succession of simulations within a single lesson. It was also redesigned to support marking sub-lessons as complete and suitable for purging from memory. These features permit the development of large lessons that need not impose very large memory demands on the student's computer at run time.

#### Phase 2: The BEESIM II Project

In the last six months of this project, an experimental extension of the iRides system was undertaken in an attempt to implement a new set of lessons about circuit theory for a Basic Electricity and Electronics course. The original plan for the USC effort on BEESIM2 was that a subset of the required simulated equipment lessons would be developed using a General Analysis product called ReAct. The project was being coordinated out of NAWC/TSD, coordinating the efforts of the original third party BEESIM authoring team, the pedagogy consultant Henry Halff, and the USC team. Our NAWC/TSD project coordinator determined that a higher priority was to come up with a new, more dynamic approach to teaching circuit behaviors in the context of simulations. (The third party authoring team undertook the completion of the required simulated equipment lessons.) The design of this new software system was developed by Henry Halff (2001). The goal of this design was to provide effective web-delivered lessons on electric circuits. These lessons were to be driven by authored specifications of several types, and were to integrate simulation and instruction in an effective way.

Halff's plan called for the use of a progression of models of circuits. It employed both qualitative and quantitative reasoning approaches, and it sought to contextualize learning in realistic circuit contexts. The student interface was to provide a number of tools to

support self-directed learning and self-help in the context of assigned tasks. Four major types of task environments were envisioned.

- Directed exploration in the context of breadboard simulations
- Interactive visualizations for explaining or clarifying concepts
- Working with on-line technical manuals
- Applications—Answering questions about real devices that exemplify the characteristics of the illustrated circuits

A variety of activities were to be assigned in these learning contexts.

Specification based training is accomplished by using a delivery system that can read the specifications and deliver interactive training based on them. The delivery system envisioned by Halff could not be the standard iRides system alone, because the specification formats he prescribed were not simply the two specification formats of iRides (the simulation language format and the LML training specifications).

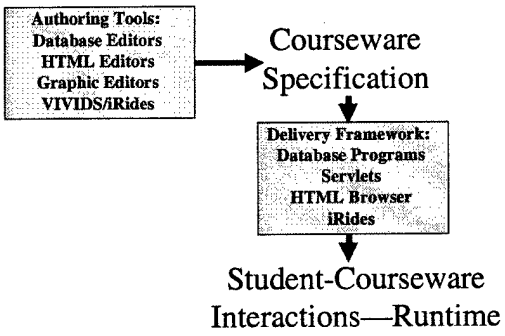


*Figure 10. Several Types of Specifications*

The many specification formats of BEESIM2 were to be these.

| *Format* | *Uses in BEESIM2* |
|---|---|
| iRides simulation files (.jr) | Breadboard simulations<br>Interactive visualizations<br>Certain applications |
| HTML Forms | Structure of problem sets |
| Problem Data (Type of database table) | Content of problem sets |
| Exercise control specifications (in LML) | Conduct (presenting, judging, remediating) of exercises |
| Exercise Data (Type of database table) | Content of exercises |
| Prerequisite Network Data (Type of database table) | Adaptive lesson selection presentation |
| Class enrollment (database table) | Login interface |
| | |

As can be seen from the list of learning contexts shown in the *Uses* column of the above table, many delivery mechanisms had to be developed to realize the Halff design. Once these delivery mechanisms were developed and tested, the databases and other specifications could be populated, and the BEESIM2 trainer would be a reality.

## Delivery Mechanisms for BEESIM2[6]

[6] This section is adapted from Halff (2001, personal communication).

### Adaptive lesson selection

The delivery of the adaptive lesson selection view was accomplished by creating a dynamic web page driven by servlets, written in Java, resident on the web server. The servlet code accesses two kinds of data: the lesson prerequisite table and the record of this student's experience with each lesson. Clicking on a lesson name causes an HTML lesson preview page to be shown in the frame at the right. The data about which page to show is also stored in a database table and is accessed by the servlet that constitutes the delivery mechanism for this user interface. The student begins the selected lesson by clicking on a Start button below the lesson preview frame.

### The Breadboard Circuit View

One of the servlets composes the page for the Breadboard Circuit View. The frame includes the version of the simulation that most closely represents the circuit being dealt with. As we see in Figure 11, the breadboard view can contain fairly realistic representations of a switch, a light, a battery, and a multimeter. The student can manipulate the various parts of the breadboard simulation and immediately see the results. In the top right corner of the frame is some general text describing what is being represented and what can be done with the simulation. To the left of that text, is a box of text that is more specific. It might discuss a particular aspect of the simulation, such as what you can expect to see if you perform various manipulations. Finally, in the lower right corner of the frame is an exercise that requires the student to respond to various questions, usually after performing some operation with the simulation. The table in this section is a new feature developed with this contract. A box with a white background indicates that the simulation has at some time since this exercise began been in a state that would provide the student with enough data to answer the question related to that box. In the example shown, the box represents the question "When the Switch Position is Open, what is the state of the Light?". As the student manipulates the switch and the multimeter, other boxes become white, indicating that the student should be able to answer that portion of the question. When a student selects a white box, an interface opens for the student to enter his or her answer to that box's question, and that answer gets inserted into the box. When all boxes have been filled, the student clicks the 'Enter' button (not shown in this figure) to signify that he/she is ready to be evaluated. The instructional code underlying this table question then evaluates the table as a whole, and reports whether enough boxes have been filled in correctly.

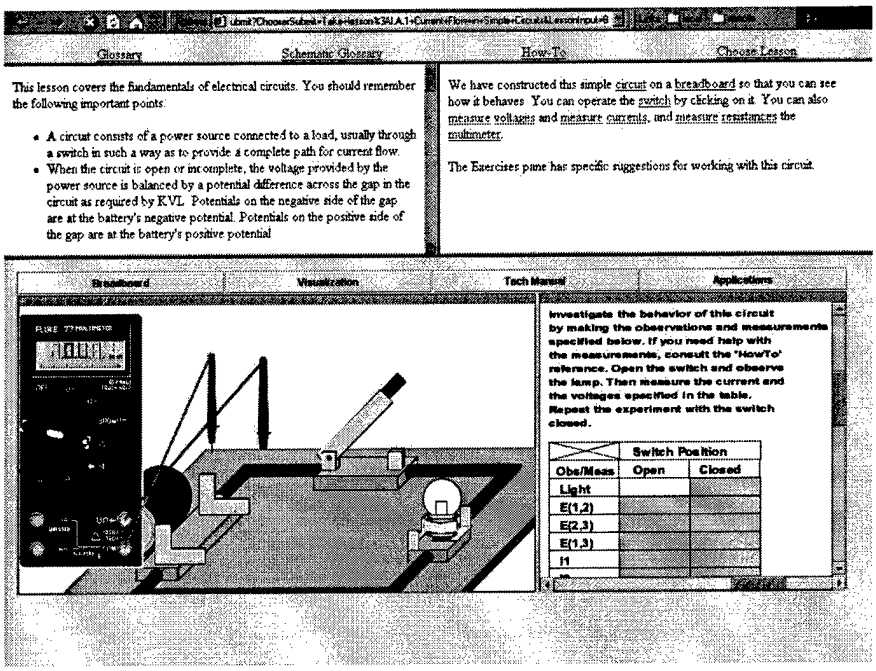


*Figure 11. The Breadboard Circuit Interface*

### The Visualization View

The same servlet that composes the Breadboard Circuit View also composes the Visualization View. This is done when a 'Visualization' button is selected. Two text boxes at the top of the window serve the same basic function that they do for the Breadboard Circuit View; the right hand one gives general information about the view, and the left hand one gives more specific information, depending on how far along the student is in the exercise. The bottom left box contains an iRides simulation, but this one is less realistic and more abstract. Also it can include pieces that don't exist in the real world, but which can contribute to student understanding by giving the user opportunities to interact with a simulation-based visualization. An exercise is provided to the student that the student can complete by carrying out mini-experiments in the simulation view.

### Problem Set Lessons

A Problem Set Lesson consists of a set of questions that the student is expected to answer. In the problem set lessons there are three types of problem questions: "Fill-in" (text and numeric), "radio buttons", and "check box". The problem data is expressed as a plain text (ascii) file, and is formatted such that each question is contained in a "*Question Definition Section*" consisting of a "*Question Definition Line*" followed by a number of varying format "*Sub-Question Definition*" lines, one such line per sub-question. In addition, a "hinting" mechanism is supported. Problem Set lessons are handled with servlet-based Dynamic HTML, and were implemented without using any of the standard iRides components.

### The Adaptive Glossary

As the student moves through instructional material, he/she is introduced to various terms. The Adaptive Glossary is a current snapshot of terms introduced in the current lesson as well as all terms introduced in all previously visited lessons. The glossary is comprised of three main components. The Lesson T erms view (upper left view of Figure 12, labeled "Lesson Terms"), the All Terms view (lower left of Figure 12, labeled "All Terms"), and the Term Definition view (center view of Figure 12). The glossary is comprised of HTML, created dynamically and delivered via a Java servlet embedded within a web server. The servlet constructs the page through a series of interactions with a relational database containing links to glossary definition text, glossary-lesson relationships, and current and past lessons visited by the student. When redrawn, the glossary servlet queries the database for the current student and lesson. With the current lesson, the servlet then queries the database for a list of all terms introduced by that lesson. The Lesson Terms view is created from this list. To build the All Terms view, the servlet queries the database for a list of all previously visited lessons by the current student, and then this list is used in another query to determine all previously introduced glossary terms. Clicking on a term from either term view initiates the creation and display of the last view, the Term Definition view. The servlet, notified of the request to view a particular term queries the database, using the selected term, for a link to the HTML page containing the desired term definition.

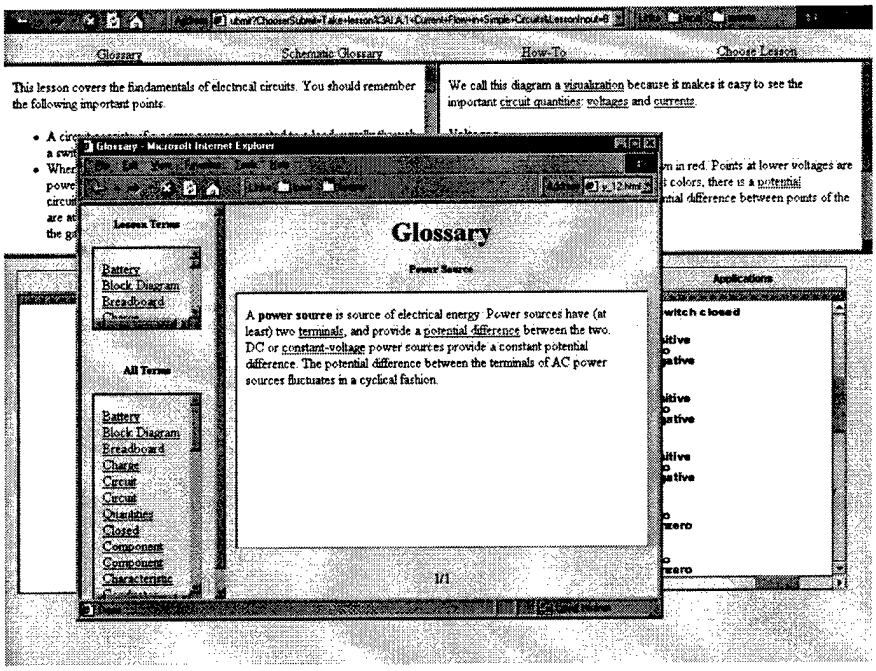


*Figure 12. The Adaptive Glossary Interface*

### The Procedure Glossary

In addition to introducing terms, lessons introduce various procedures. The Procedure Glossary is a current view of all procedures introduced up to the time the glossary is

redisplayed. The glossary is comprised of HTML, created dynamically and delivered via a Java Servlet embedded within a web server. The Procedure Glossary is very similar in design and functionality to the Adaptive Glossary, with the difference being the nature of the data residing in the relational database.

### The Circuit Simulations

The set of progressively more complex circuits to be used in Halff's design can be viewed as instances of one complex circuit in particular configurations. Rather than author each of these simulations independently, we developed a single simulation, shown below in Figure 13, that comprises the behavior of all the others. An authoring interface was developed, using the iRides simulation language, to let authors quickly build and save each of the required simulation variants. When authors hold down the mouse on any active component in this circuit, a popup menu appears that allows them to select what type of component should appear at that point. In this way, an author can quickly build a variant of the circuit and test it for use in a lesson.

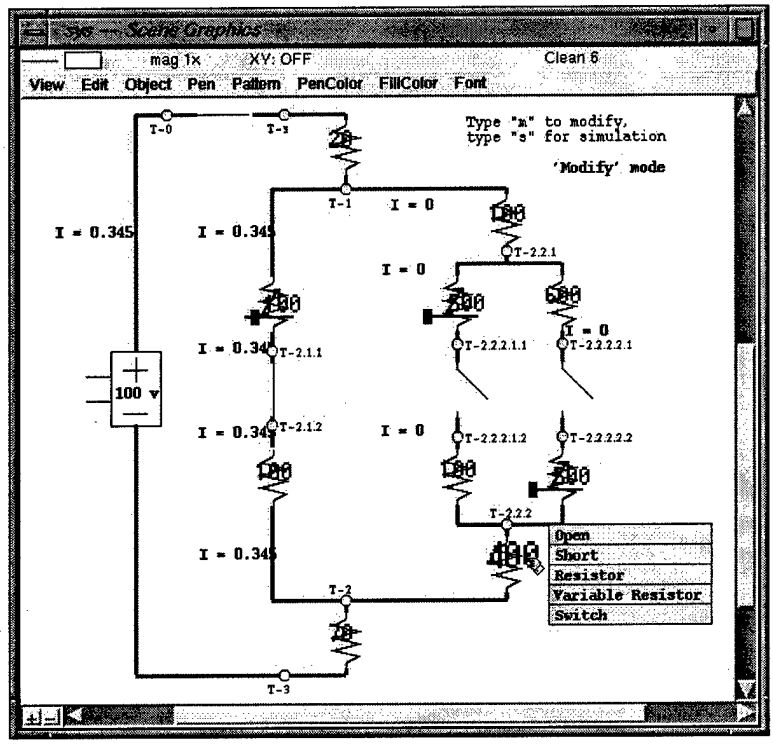


*Figure 13. The Circuit-authoring Interface*

The behavior of breadboard circuits and visualizations can be driven from this underlying circuit simulation. That is, the values that are computed for current and voltage in this simulation can be observed by the simulation views that students actually see.

### Results of the BEESIM2 Effort

During the course of this project, all the data formats were designed, and all the major delivery mechanisms were developed and tested. Two small lessons were developed, a

circuit lesson and a problem set. The major mechanisms are now in place for the delivery of circuit theory lesson content based on authored specification. Significant efforts would be required, however, to develop all the planned content that could be delivered by this system.

## Papers Resulting from this Grant

Munro, A., Surmon, D., Johnson, M., Pizzini, Q., and Walker, J. (1999) An Open Architecture for Simulation-Centered Tutors. In Lajoie, S. P. and Vivet, M., *Artificial Intelligence in Education: Open Learning Environments: New Computational Technologies to Support Learning, Exploration, and Collaboration.* Amsterdam: IOS Press, 360-367.

Munro, A., Breaux, R., Patrey, J. and Sheldon, B. (2002) Cognitive aspects of virtual environments design. In Stanney, K. (Ed.), *Handbook of Virtual Environments*. Mahwah, New Jersey: Lawrence Erlbaum Associates.

Munro, A. (in press) Teaching in the context of authored simulations. In Murray, T. (Ed.) Authoring Tools for Advanced Technology Learning Environments: Toward cost-effective adaptive, interactive, and intelligent educational software. Dordrecht: Kluwer.

## Continued iRides Development

### Status of the Software

iRides development has continued since the end of the *Intranet Delivery of Simulation-Centered Training* project. This work is supported by the Future Navy Capability (FNC) program through grant N00014-02-0179 to the Center for Research on Evaluation, Standards, and Student Testing (CRESST) at UCLA. CRESST, in turn, has issued award 0070-G-CH640 to the University of Southern California. Many of the lessons of the project covered in this final report have helped to guide continued iRides development under FNC funding.

#### Completion of the iRides Debugger.

The new iRides debugger completes the design and preliminary implementation begun in the Intranet Delivered Simulation project. It is used by authors to understand why a simulation is not behaving as expected, perhaps due to a mistyped rule or due to an interaction of rules not foreseen by the author. The debug editor can be used to examine the attribute constraints just as they are about to be evaluated and event statements just as they are about to be executed. This interface is often used in conjunction with paused attributes and with the simulator's Step capability, which steps through rules at the direction of the author. Also the author can use the Step In and Step Out capabilities for greater control over the debugging of event bodies. If the author does not want to dive into a particular event body, the Step command is used. If the author wants to pause at each line of the next event body, the Step In feature can be used; if no further lines within the body of that event are of interest, the Step Out feature is used. Using the debug editor together with pause and/or the Step command, the author can

- observe the name and the structure of the next rule to be run
- select and evaluate expressions within the body of the rule to be run next
- observe the list of names of attributes and events marked as paused
- specify which attributes or events to stop for when stepping. It is usually the case that only a few attributes or events are 'interesting' when debugging, and it is a waste of effort for the debug editor to pause for other items when stepping through the simulation.

The simulation parser's treatment of comments in defined simulation events had to be modified so that the parser's line numbering could be used by the debugger's pretty-printing routines. This makes it possible to show the text of the event, including the comments, and to accurately point in the debugger to each statement as it is executed in turn by Stepping. In Figure 14, the pointing hand icon at the left indicates which line of the iRides Event will be carried out next, when the Step button is pressed. Clicking on Step Out would complete the entire event without stepping through the remaining statements individually. The debugger can handle nested DoEvents (that is, nested function calls in the iRides simulation language). In addition, pretty-print routines display the simulation language with formatting that enhances the user's understanding of the structure of the simulations.

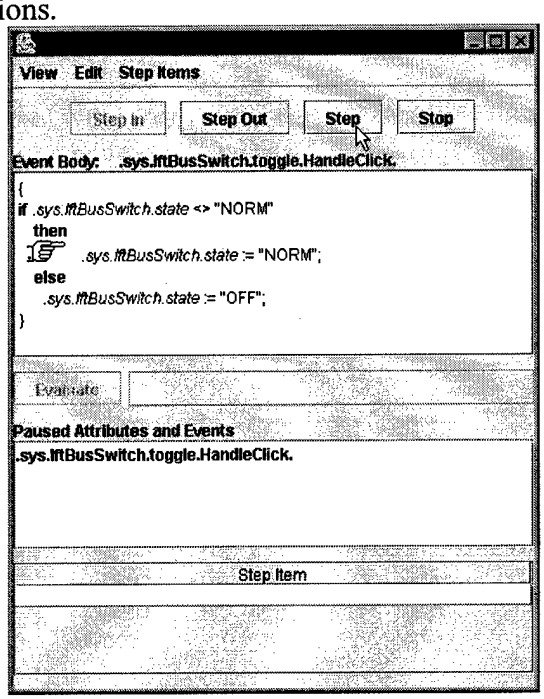


*Figure 14. Debugging Window with Formatted Simulation Rules*

## Scalable lesson/course structures.

Challenges encountered in the VIVIDS-based BEESIM course structures helped to motivate a more scalable approach to specifying course and lesson structure. In current (September 2003) iRides, any lesson can incorporate any other lesson by reference. That makes it possible to create complex hierarchical lesson/course structures with multiple

layers. Only two levels—lessons, which could have only one simulation, and courses, which could refer to multiple lessons, but could not contain other courses. Only two levels of pedagogical grouping were possible. Any level of pedagogical nesting is possible in the new iRides. That gives instructional designers more flexibility, and they don't have to learn about two completely different types of specifications: that for courses and that for lessons.

### Improved LML file structure to limit size and improve readability.

As described above in *Results and Discussion*, the experience of working with the automatically exported LML lesson specifications from the BEESIM lessons pointed out the need for a more succinct representation that was easier to read and make sense of. A recent implementation has made good use of the iRides construct *lesson templates*. In this approach, an LML lesson file only needs to specify certain information about each lesson segment. During lesson delivery, the training controller uses the file together with separate, standardized files that specify all the invariant details. As an example, the code for a 'find object' lesson segment in VIVIDS needs to know what object the student is supposed to find, but unless the author has specified otherwise, assorted output texts are generated automatically, such as 'No, that is not correct. Try again.' or 'You have run out of time.' The initial LML conversion files could be as much as 50 lines long just to handle each simple Find Object item. However, with the use of lesson templates, the same result can be achieved with as few as three lines:

```
<use href="Tfind.lml#VFind">
   <setProperty name="Object" value=".Front_Panel.Power_Light."/>
</use>
```

The first line specifies the name of the template file that is to be used. The template file takes care of the looping—repetitions in case that student's first attempt is incorrect—and all the text that will be displayed in various situations. The second line specifies the object that is to be found. And the third line is just to close off the 'use' item. If the author had chosen to display text other that the standard boilerplate found in the template, (s)he could do so in the VIVIDS lesson, and this would result in additional 'setProperty' lines in the LML file. By incorporating lesson templates into the iRides files dumped by VIVIDS, those iRides files, in addition to becoming much shorter, became much more readable.

### iRides Availability

For information about accessing the iRides software and examples of iRides simulations and lessons, contact Allen Munro, munro@usc.edu.

# References


Cunningham, R. E., Corbett, J. D. Bonar, J. G., 1987. *Chips: a tool for developing software interfaces interactively*. Pittsburgh: Learning Research and Development Center, University of Pittsburgh.

Forbus, K., 1984. *An interactive laboratory for teaching control system concepts*. (Tech. Report 5511). Cambridge, Massachusetts: Bolt Beranek and Newman Inc.

Halff, H., 2001, *Design Framework for BEESIM Circuit Lessons*. Personal Communication.

Hollan, J. D., Hutchins, E. L., & Weitzman, L., 1984. STEAMER: An Interactive Inspectable Simulation-based Training System, *AI Magazine*, **2**.

Ingalls, D., Wallace, S., Chow, Y., Ludolph, F., & Doyle, K., 1988. Fabrik, a visual programming environment. *Proceedings OOPSLA '88*. New York: ACM, 176-190.

Johnson, W.L. and J. Rickel, 1996. "Intelligent Tutoring in Virtual Environment Simulations," ITS '96 Workshop on Simulation-Based Training Technology.

Johnson, W. L., Rickel, J., Stiles, R. and Munro, A., 1998. Integrating Pedagogical Agents into Virtual Environments. *Presence*.

Leler, W., 1988. *Constraint programming languages: their specification and generation*. Menlo Park, CA: Addison-Wesley.

Linton, M. A., Vlissides, J. M., & Calder, P. R. , 1989. Composing user interfaces with InterViews. *Computer* 22(2), 8-22.

Munro, A., 1994. Authoring interactive graphical models. In T. de Jong, D. M. Towne, and H. Spada (Eds.), *The Use of Computer Models for Explication, Analysis and Experiential Learning*. Springer Verlag.

Munro, A. (in press) Teaching in the context of authored simulations. In Murray, T. (Ed.) *Authoring tools for advanced technology learning environments: Toward cost-effective adaptive, interactive, and intelligent educational software*. Dordrecht: Kluwer.

Munro, A., Breaux, R., Patrey, J. and Sheldon, B. 2002. Cognitive aspects of virtual environments design. In Stanney, K. (Ed.), *Handbook of Virtual Environments*. Mahwah, New Jersey: Lawrence Erlbaum Associates.

Munro, A., Johnson, M. C., Pizzini, Q. A., Surmon, D. S., Towne, D. M. and Wogulis, J. L, 1997. Authoring Simulation-Centered Tutors with RIDES. International Journal of Artificial Intelligence in Education, **8**, 284-316.

Munro, A. Johnson, M.C., Surmon, D. S., and Wogulis, J. L., 1993. Attribute-centered simulation authoring for instruction. In the Proceedings of AI-ED '93—World Conference on Artificial Intelligence in Education.

Munro, A. and Pizzini, Q. A. , 1998. *VIVIDS Reference Manual*, Los Angeles: Behavioral Technology Laboratories, University of Southern California.

Munro, A., Surmon, D., Johnson, M., Pizzini, Q., and Walker, J., 1999. An Open Architecture for Simulation-Centered Tutors. In Lajoie, S. P. and Vivet, M., *Artificial Intelligence in Education: Open Learning Environments: New Computational Technologies to Support Learning, Exploration, and Collaboration*. Amsterdam: IOS Press, 360-367.

Stiles, R., McCarthy, L., Munro, A., Pizzini, Q., Johnson, L., and Rickel, J., 1996. *Virtual Environments for Shipboard Training*, Intelligent Ship Symposium, American Society of Naval Engineers, Pittsburgh PA.

Stiles, R., McCarthy, L., and Pontecorvo, M., 1995. Training studio: a virtual environment for training, Workshop on Simulation and Interaction in Virtual Environments (SIVE95) Iowa City, IW: ACM Press.

Sutherland, I. E., 1963. Sketchpad: a man-machine graphical communication system. Proceedings of the Spring Joint Computer Conference, 329-346.

Towne, D. M.,1977. Approximate reasoning techniques for intelligent diagnostic instruction, *International Journal of Artificial Intelligence and Education.*

Towne, D. M., 1998. *DIAG: Diagnostic instruction and guidance*. Los Angeles: Behavioral Technology Laboratories, University of Southern California.

Towne, D. M., 1998*DIAG: Diagnostic instruction and guidance—application guide, v. 2*..Los Angeles: Behavioral Technology Laboratories, University of Southern California.

Vlissides, J. M. , 1990 Generalized graphical object editing. Doctoral dissertation, Stanford University.

Wilde, N. & Lewis, C., 1990. Spreadsheet-based interactive graphics: from prototype to tool. In *Proceedings CHI '90*, New York: Association for Computing Machinery, 153-159.