John Domingue and
Paul Mulholland

Fostering Debugging Communities on the Web

Programmers worldwide can collaboratively debug programs synchronously and asynchronously by tapping into the Internet Software Visualization Laboratory.

HOW OFTEN DO WE GO TO A COLLEAGUE or local software guru to solve a seemingly intractable bug? Frequently such people can locate in minutes a bug we've spent days trying to track down. There is evidence (see Eisenstadt's article in this issue) that the mere act of explaining a bug to a colleague speeds the bug-tracking process. However, the programmers we'd most like to talk to are often not available. Or maybe there is no local guru or anyone using the same language or hardware platform as the one with the bug. Although one might think the Internet would be useful for finding help to eradicate bugs, so far it is underutilized. That is why we

describe our approach to fostering debugging communities on the World-Wide Web, enabling programmers to collaboratively debug programs synchronously and asynchronously.

Over the past couple of years, a significant number of communities have benefited by interacting through the net using newsgroups and Web pages. Net-based debugging communities would profit from:

- Free help 24 hours a day
- Potential access to world-class experts (e.g., the people who created the language)
- Help for niche areas where a language or particular configuration is not widely used

Although newsgroups exist for most of the popular computer languages, they tend to discuss general issues and problems rather than specific bugs. Here, we discuss the reasons for the lack of net-based debugging communities, our approach illustrated by a limited scenario, related work, and our future directions.

Why the Lack of Debugging Communities?

Given the potential benefits for programmers, it may be surprising that there are no net-based communities for describing and fixing bugs. The reason is that, using current technology, the effort needed to describe a bug and to understand someone else's bug is too great.

To describe a bug, a programmer needs to describe the code, the point in the execution where the symptom occurs, and the context. The context includes information necessary for replicating or understanding the bug not directly contained within the code (e.g., Eisenstadt in this issue describes a program that works only on Wednesdays). Today, programmers are limited to impoverished descriptions in plain ASCII text. One way of alleviating some of these problems is by sharing code, but this is not easy. Running another programmer's code can be prevented by differences in:

- Platform. There are many types of hardware and operating systems.
- Language. Many computer languages have different dialects or syntactic variants. Some languages are not fully specified, and implementations may have differences in, say, they way they handle graphics or foreign function calls.
- Version. The local version of required software may be out of date, or the shared code may require a feature from an older version.

The Laboratory uses a client/server architecture to deliver visualizations on any Java-enabled Web browser.

- Libraries. The code may require installation of a particular library.
- Configuration. The code may require a particular system configuration.
- CPU speed and memory requirements. A machine may lack adequate performance to run the code in a reasonable time or even to run the code at all.
- Commercial interests. The program may require commercially sensitive modules that cannot be released.

OUR AIM IS TO MAKE IT easy for programmers to swap bug descriptions around the world. Therefore, our system has to be platform-independent, run on relatively modest hardware, and not require exchange of source code, high bandwidth, or synchronous communication. Addressing these points, we need to consider two main issues: What exactly will be shared and what mechanisms are required to support the sharing process. For the first issue, we need to think about what a bug description is and what the debugging process involves. A bug description is essentially a description of what the program did and what the program should have done. The debugging process involves two main tasks: The first is to find the mismatches between the descriptions of what the program did and what it should have done; the second is to locate the points in the source code corresponding to the mismatches. As eloquently argued in Lieberman's introduction in this issue, software visualization (SV) techniques can aid in displaying a program's execution. Our approach to what the program should have done is to allow programmers to annotate their visualizations using simple drawing and labeling tools.

How best to support the sharing process? If we can encode our annotated visualizations as HTML files and Java code, Web and Java technology will enable us to share these without any of the program-difference problems. The remaining problem is to provide a framework facilitating generation of visualizations encoded as HTML and Java.

Internet Software Visualization Laboratory

Our approach to solving the bug-description-sharing problem is the Internet Software Visualization Laboratory (ISVL), which uses a client/server architecture to deliver visualizations on any Java-enabled Web browser (see Figure 1).

Using a Java-enabled Web browser, programmers can connect to the ISVL server and download the ISVL client. Using the client, programmers can upload and run their programs on the server and receive back a visualization. Programmers can also use the client to collaboratively debug programs. (An example of a collaborative debugging session is described in the following section.)

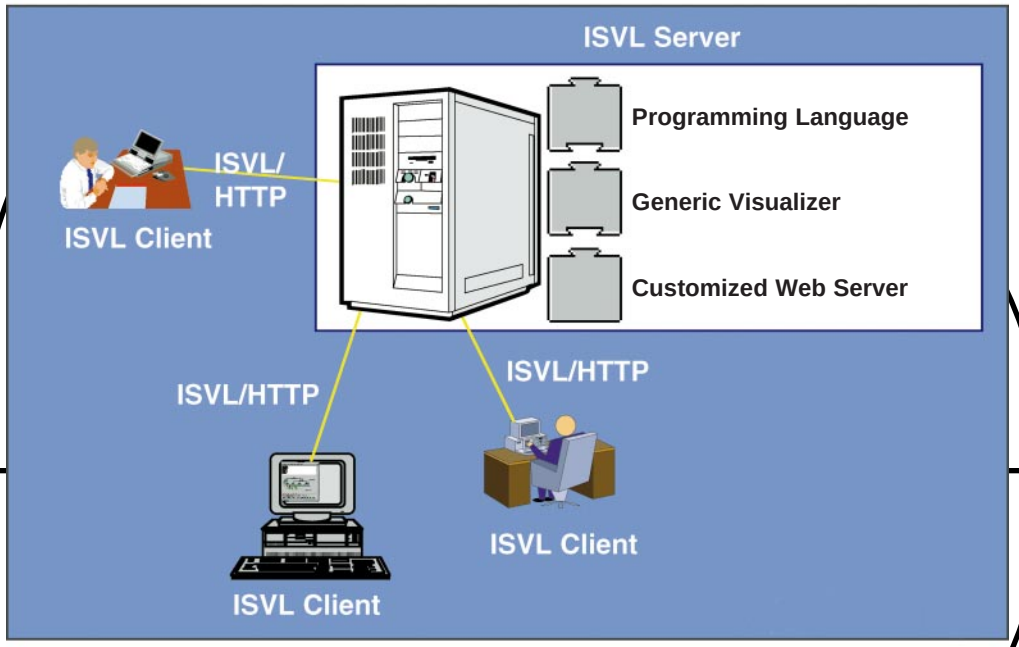

Figure 1. Architecture of the ISVL

The ISVL server is composed of a customized Web server, a generic software visualizer, and a specific programming language. The customized Web server is based on LispWeb [10], a specialized HTTP server written in Common Lisp. In addition to implementing the standard HTTP protocol, the LispWeb server offers a library of high-level Lisp functions for dynamically generating HTML pages, a facility for dynamically creating image maps, and a server-to-server communication method.

The generic visualizer is an extension of our framework—called Viz—for creating SVs [5]. Within Viz, we view program execution as a series of history events happening to (or perpetrated by) players, that is, Viz abstractions for computational elements that can change state. To allow SV system builders considerable freedom, a player can be any part of a program, such as a function, a data structure, or a line of code. Each player has a name and is in a state that may change when history events occur for that player. A player may also contain other players, enabling the formation of groups of players. History events are like Brown's [3] "interesting events" in the Brown University Algorithm Simulator and Animator (BALSA)—an environment for creating and viewing algorithm animations in which each event corresponds to code being executed in the program or data changing its value. These events are recorded in the history module, allowing them to be accessed by the user and "replayed" later. Events and states are mapped into a visual representation accessible to the end user, that is, the programmer needing to use the SV system, not the SV system builder. But the mapping is not just a question of storing pixel patterns to correspond to different events and states; we also need to specify different views and ways of navigating around them. The main Viz ingredients are:

- Histories. A record of key events occurring over time as the program runs, with each event belonging to a player. Each event is linked to some part of the code and may cause a player to change its state. Also included is some prehistory information before the program begins running, such as the static program source code hierarchy and initial player states.
- Views. The style in which a particular set of players, states, or events is presented. Examples are text, a tree, or a plotted graph, each using its own style and emphasizing a particular dimension of the data it displays.
- Mappings. The encodings used by a player to show its state changes in diagrammatic or textual form in a view using a graphical language, typography, or sound. Some of a player's mappings may be for the exclusive use of its navigators.
- Navigators. The interface tools that enable the

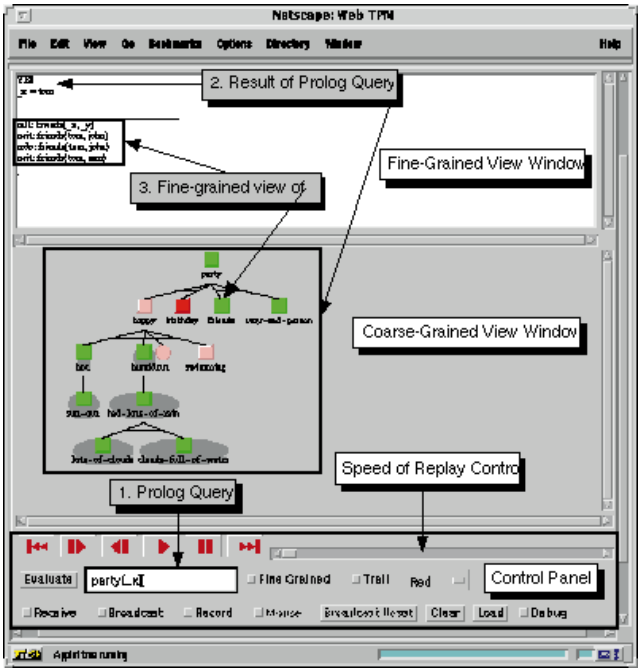

Figure 2. The ISVL Prolog client

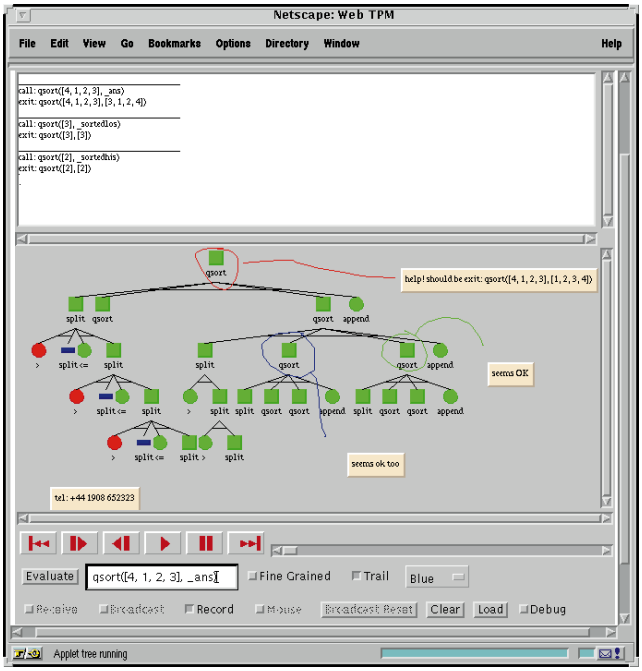

Figure 3. A snapshot from the movie left by Bill

user to traverse a view, move among multiple views, change scale, compress or expand objects, and move forward or backward in time through the histories.

The Viz framework is equally at home dealing with program code and with algorithms, since a player and its history events may represent anything from a low-level (program code) abstraction, such as `Invoke a function call`, to a high-level (algorithm) abstraction, such as `Insert a pointer into a hash table`. The mapping module in Viz is interfaced with LispWeb so plain ASCII representations are sent to the client, thus reducing required bandwidth.

A programming language is interfaced to Viz by inserting `Create player` and `Note interesting event` hooks. To date, we have created more than a dozen visualizations using Viz (see [5]).

On the client side, a transformation module converts the ISVL HTTP stream into textual and graphical representations that are then transformed and presented on the screen by the navigator. The user interacts with the visualization through the navigator, which controls panning, zooming, local compression, expansion, and moving forward and backward in time through the program execution space.

The First System

Our first system—called Web-TPM—is being used and evaluated in the context of an Internet, and we are running an Internet version of our master's-level Intensive Prolog course [7] using a visualization based on the Transparent Prolog Machine (TPM) [6]. Here we describe an interaction between two hypothetical programmers, Bill and Ingrid. Because the purpose of the scenario is to show how ISVL supports SV-based collaborative debugging, not to show off the debugger itself, we use a trivial program. Please note that ISVL contains many features allowing it to scale up to cope with arbitrarily large programs.

Figure 2 includes labels describing the parts of the ISVL Prolog client. A user types a query in the Prolog Query window (1). The result of the query `No` (indicating that the query failed) and a TPM-style visualization are returned (2). The user can then step through the execution using the move-right button in the control panel. A fine-grained view of the node can be obtained by clicking on the node. Figure 2 shows a fine-grained view of the node "friends" (3).

In our scenario, Bill is writing a sorting program called Qsort, sorting a number of unsorted elements, based on the Quicksort algorithm, which splits a list around an element into a list of lower numbers and a list of higher numbers that are then recursively sorted. The program should take an unsorted list and return a sorted list. Bill's current version is buggy, and he cannot find the root cause. He starts recording a movie by selecting the `Record` button in the control

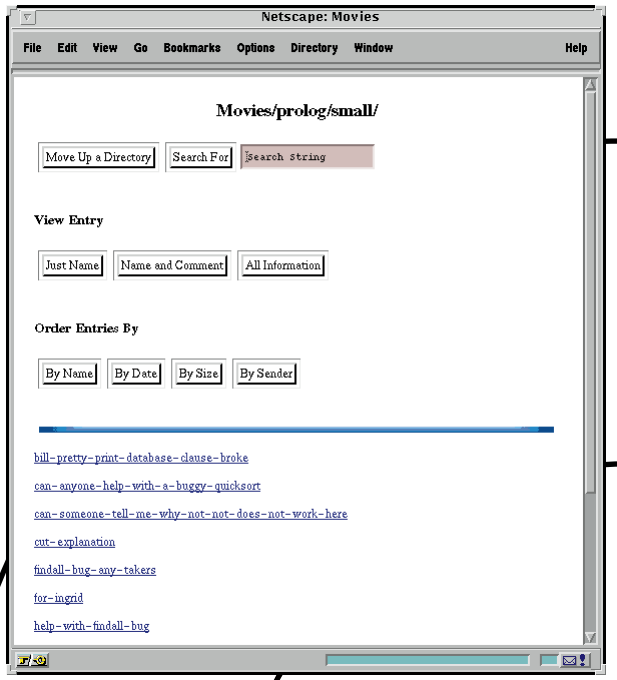

Figure 4. The ISVL movie database Web page

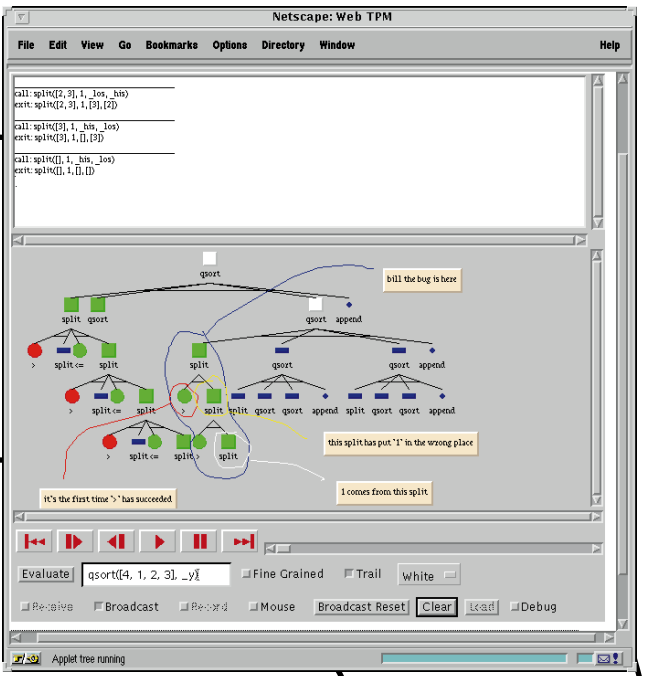

Figure 5. Ingrid's ISVL client after a collaborative session with Bill

panel, then carries out the following steps:

- Clicks on the topmost Qsort node, causing the fine-grained information:

```
call: qsort([4, 1, 2, 3], _ans)
exit: qsort([4, 1, 2, 3], [3, 1,
 2, 4])
```

  to appear in the fine-grained view window.
- Rings the top node with a red circle using the mouse (the color is set using the choice button on the control panel) and inserts the top annotation `Help! should be exit: qsort([4, 1, 2, 3], [1, 2, 3, 4])`.
- Rings the lower Qsort nodes on the right with a green circle and Qsort nodes on the left with a blue circle and adds the annotations `seems OK` and `seems OK too`. The state of his screen at this stage is shown in Figure 3.
- Ends the recording session by selecting the `Record` button, and when prompted, names his movie `can-anyone-help-with-a-buggy-quicksort`.

Some time later, Ingrid looks up the Prolog movie database (see Figure 4), retrieves Bill's movie, and plays through it. The interface for playing a movie is almost exactly the same as that for viewing visualizations, except the visualization steps are primitive user actions, such as circling or labeling a node and clicking on a node, rather than steps in the program execution. She plays through the movie, checking the fine-grained views of the three split nodes in the subtree to the left of the Qsort node circled in blue (containing three generations and five nodes). While checking these split nodes, she finds the bug. Instead of leaving a movie for Bill, she calls him by phone, and they agree to start a synchronous collaborative session.

Bill puts his client into Receive mode by selecting the `Receive` button in the control panel. Ingrid puts her client into Broadcast mode by selecting the `Broadcast` button in the control panel. She loads Bill's code and runs it by typing the query into the query window and selecting `Evaluate`. Then, to put her interface into the state shown in Figure 5, she carries out the following steps:

- Hits the right-arrow button until the tree is as shown in Figure 5, then stops by using the pause button.
- Rings and annotates parts of the tree in the order blue, red, yellow, white. Each time she does so, she also brings up a fine-grained view (Figure 5, top window) of an appropriate split node.

Each interface action is replicated on Bill's client. After the session, Bill fixes his bug and checks the new code on the server.

Benefits

Because ISVL uses Java-based clients, it gains all the benefits Java brings to an application:

We expect programmers who are particularly isolated or at the bottom of a learning curve to benefit most.

- The ability to run on multiple platforms
- The ability to run on low-cost platforms (with the potential of one day running on set-top boxes and handheld devices)
- Radically decreased effort of delivery, maintenance, version control, and update shipment

To facilitate ease of use, we also incorporated other design features into ISVL:

- Minimal bandwidth requirements. Visualizations and broadcasts are encoded as plain ASCII.
- Minimal connect time. Once a visualization is delivered to a client, all interactions are local. The only time a connection is left open is when a client is in Broadcast or Receive mode.
- Minimal use of the server. Because movies are plain HTML files with accompanying Java code, they do not require the ISVL server.
- Maximum control. Movies are not just watched and seen. A programmer can control the speed of the replay and obtain fine-grained views of nodes not in the original movie.

Today, we are testing ISVL on an Internet version of our master's programming course, which involves students in the U.S. and Canada and a tutor in England. Although it is too early to draw conclusions from this experiment, our expectations are based on earlier experiments on Internet teaching [12], as well as on empirical studies of novice programmers using a variety of SV systems [9]. Any programming community spread over distance or time can benefit from our approach. Exactly how the framework is used depends on the type of community and its various dimensions, including:

- Openness. Is membership in the community available to all programmers? Are community activities available for inspection? Two communities differing greatly on this dimension are academia and the military, which requires a secure intranet.
- Size. The size of a community shapes system use.
- Homogeneity. A community may be made of distinct subcommunities, each with its own agenda. The relationships among the subcommunities are important. For example, software providers have a special relationship with their customers. Alternatively, a community may have a strong internal structure (e.g., a strictly hierarchical corporation influences communication among its employees).

The benefits for individual programmers depend on their roles within a community. We expect programmers who are particularly isolated (e.g., a consultant working at a remote client site) or who are at the bottom of a learning curve (e.g., a new member of a software team) to benefit most.

Related Work

Recent work within the algorithm-animation community has led to the creation of a number of systems allowing pre-written animations to be viewed remotely. These systems primarily concentrate on delivering animations synchronously as part of a classroom-style tutorial.

For example, John Stasko developed a facility at the Georgia Tech College of Computing (http://www.cc.gatech.edu/stasko/cgi-bin/xtangoanim) where Transition-based Animation Generation (TANGO) [11] animations can be run remotely. Although useful for quickly getting an overview of TANGO's look and feel (the facility's purpose), the system is restricted to X Window systems and has scaling problems, as all the work is carried out on a central server.

The Collaborative Active Textbooks (CAT) of Brown and Najork [4] at Digital Equipment Corp.'s Systems Research Center is a Web-based environment allowing the same animation to be run simultaneously on a number of machines. Intended for classroom-style teaching, the tutor has remote control of the view and speed of the animation. Such synchronous demonstration of programs is possible through ISVL, though animations in ISVL are not canned, being created on the fly from the program submitted.

The client/server architecture of ISVL is similar to that of James Baker's Mocha system [1] at Brown University, in which the bulk of the work is done on the server, and the interface is created by a Java client. Like CAT, Mocha is primarily designed for the synchronous delivery of algorithm animations.

Future Directions

You may now ask, what about voice? Programmers with Ethernet links and a telephone can use voice

with ISVL in a synchronous fashion; it is also possible to add links to .au files, the Unix-specific format for audio files, playable on any Java-aware Web browser, within movies. However, we feel our current setup has limitations, as some programmers use a dial-up modem (especially when at home) and .au files are large and cannot be handled well on all platforms. In addition, nonlocal phone calls are expensive. We therefore plan to use the streaming audio system currently being constructed as part of KMi Stadium [8], a Java-based application exploring the use of large-scale telepresence, particularly for broadcasting real-time audio.

In his debugging environment for Multiple Representation Environment (MRE), Mike Brayshaw [2], who co-developed TPM, described how a subpart of a visualization could be parameterized and stored away as a "symptomatic agent" that could then be used to search in the execution space of future programs. We plan to enable programmers to use direct manipulation to capture, or isolate and store, parts of their current visualization and to ask, has anyone else had a problem like this?, at which point an agent would search known movie databases for matching visualizations. In addition to investigating other languages, such as Java, we are using the framework in a wider context to support collaborative case-based engineering design and collaborative ontology browsing and editing. C

Acknowledgments

The authors would like to thank Enrico Motta and Simon Buckingham Shum for providing valuable feedback on various drafts of this article.

References

1. Baker, J.E., Cruz, I.F., Liotta, G., and Tamassia, R. Algorithm animation over the World Wide Web. In *Proceedings of the International Workshop on Advanced Visual Interfaces*. ACM Press, New York, 1996.
2. Brayshaw, M. *Information Management and Visualization for Debugging Logic Programs*. Ph.D. dissertation, Human Cognition Research Laboratory, the Open Univ., Milton Keynes, U.K., 1994.
3. Brown, M.H. *Algorithm Animation*. ACM Distinguished Dissertations. MIT Press, Cambridge, Mass., 1988.
4. Brown, M.H., and Najork, M.A. Collaborative active textbooks: A Web-based algorithm animation system for an electronic classroom. Systems Research Center Res. Rep. 142, Digital Equipment Corp., 1996.
5. Domingue, J., Price, B., and Eisenstadt, M. Viz: A framework for describing and implementing software visualization systems. In *User-Centred Requirements for Software Engineering Environments*, D. Gilmore and R. Winder, Eds. Springer-Verlag, Berlin Heidelberg, 1992, pp. 197–212.
6. Eisenstadt, M., and Brayshaw, M. The Transparent Prolog Machine (TPM): An execution model and graphical debugger for logic programming. *J. Logic Program.* 5, 4 (1988), 277–342.
7. Eisenstadt, M., Dixon, M., and Kriwaczek, F. *Intensive Prolog*. Academic Press, Milton Keynes, U.K., 1988.
8. Eisenstadt, M., Buckingham Shum, S., and Freeman, A. KMi Stadium: Web-based Audio/Visual Interaction as Reusable Organisational Expertise. In *Proceedings of The Workshop on Knowledge Media for Improving Organisational Expertise*. (1st International Conference on Practical Aspects of Knowledge Management, Basel, Switzerland, Oct. 1996). Unpublished, but available from A. Nicolet, Schwandenholzstr. 286 CH-8046, Zurich, Switzerland. (Also available as Knowledge Media Institute Tech. Rep. 31, http://kmi.open.ac.uk/techreports/kmi-tr-list.html. See also http://kmi.open.ac.uk/stadium.)
9. Mulholland, P. A principled approach to the evaluation of SV: A case-study in Prolog. In *Software Visualization: Programming as a Multi-Media Experience*, J. Stasko, J. Domingue, M. Brown, and B. Price, Eds. MIT Press, Cambridge Mass., in press.
10. Riva, A., and Ramoni, M. LispWeb: A specialised HTTP server for distributed AI applications. *Comput. Networks ISDN Syst. 28* (May 1996), 953–961.
11. Stasko, J.T. The Path-Transition Paradigm: A practical methodology for adding animations to program interfaces. *J. Visual Lang. Comput. 1*, 3 (Sept. 1990), 213–236.
12. Watt, S.N.K. Teaching through electronic mail. Tech. Rep. 15, Knowledge Media Institute, Sept. 1995 (see http://kmi.open.ac.uk/techreports/kmi-tr-list.html.)

John Domingue (J.B.Domingue@open.ac.uk.) is a research fellow in the Knowledge Media Institute at The Open University in Milton Keynes, U.K.

Paul Mulholland (P.Mulholland@open.ac.uk.) is a research fellow in the Knowledge Media Institute at The Open University.