A1: Spreadsheet-based Scripting for Developing Web Tools

Eben M. Haber, Eser Kandogan, Allen Cypher, Paul P. Maglio, and Rob Barrett
– IBM Almaden Research Center

ABSTRACT

A1 is a Java-based spreadsheet environment that enables system administrators to builA1 is a Java-based spreadsheet environment that enables system administrators to build small tools that simplify and automate common tasks, integrating real-time data across heterogeneous systems. A1 spreadsheets can be saved to a central repository, where they are published and shared as interactive web portlets. In this paper, we discuss the need for administrators to create their own tools, how the A1 environment is designed to support this need, and how A1's support for web publishing-without requiring special web programming-can enable teams to share, modify, and improve their tools. We also discuss the design and implementation of A1, and show a number of sample spreadsheets for various administration tasks.

Introduction

System administration frequently involves the development of custom scripts to manage the unique requirements of each site. Typically, these scripts provide functionality lacking in commercial software tools because vendors cannot anticipate every task for every possible configuration in their tools. Custom-built scripts range from a few lines of code developed by a single administrator to speed a commonly executed task, to significant software systems developed by a team of people over several years.

Collaboration is a common aspect of system administration work. At most sites, expertise is distributed across teams who work together to design, implement, and maintain systems, and to troubleshoot problems. System administrators often share scripts as they collaborate on a task, or they pass scripts to others working on similar tasks. Scripts are often modified, customized, and improved, as they pass from person to person.

In the course of system administration field studies, we have observed several difficulties in existing scripting practices. Coordinated script use between different administrators can be hindered by a lack of central script repositories, inconsistent script versions, and varying execution environments. Web-based scripts provide consistent access and execution, but are difficult to develop. Many administrators lack the skills needed to field web applications, such as cgi scripting or J2EE servlet programming. Finally, we have seen many cases where the complexity of common scripting languages (e.g., Perl, Python, and shell scripts) prevented administrators from modifying and reusing existing scripts for their own purposes.

Though sometimes these issues result from limited programming experience, more frequently script sharing problems are due to insufficient documentation, hidden assumptions and hard-coded constants. All too often administrators who need a certain tool "re-invent the wheel," rewriting it themselves instead of building on existing scripts.

We developed A1 to enable system administrators to quickly develop custom web-based tools and to more easily share, reuse, and improve tools collaboratively. A1 is a spreadsheet-like environment that allows system administrators to invoke existing scripts or access remote systems via standard protocols, such as SSH, JMX and SNMP. A1 includes a task-specific scripting language so that connections to servers can be opened in a spreadsheet cell, and server actions can be triggered by changes to cell values. Sysadmins can save spreadsheets to a shared repository and then execute them from a web browser as interactive web portlets. A1 also provides a plug-in architecture to add support for new components.

A1 is not meant to replace existing scripting languages, such as Perl and Python. The power and flexibility of these languages will continue to be important for advanced users and experienced programmers creating complex tools. Rather, A1 is intended to provide an environment for a broader population of system administrators to create small tools that can be quickly and easily deployed and shared via the web. A1 spreadsheets can be also be used as a control layer to execute and manage the output of existing scripts developed in other languages. In summary, system administrators using A1 can build and share spreadsheets to access remote, heterogeneous systems, gather and integrate real-time data, and control various systems uniformly through a web-interface.

Background

Over the past three years, we conducted a series of field studies of system administration work. The study sites included large enterprise, university, and

government research environments. We made fourteen visits to six different sites, examining web hosting, database, operating system, security, and storage administration. We interviewed and surveyed system administrators about their work. We recorded a total of 50 days of video following normal, day-to-day activities. The results are primarily qualitative, providing a picture of typical work practices, tools, and problems faced by system administrators.

In these field studies, we made the following observations relevant to administrators' use of scripting and tools:

1) Collaboration is important for system administrators. Because no single individual can be an expert in every aspect of a large installation, responsibility is typically spread across people and organizations. During complex configuration or troubleshooting, collaboration is essential to a successful resolution. For example, in a detailed analysis of one lengthy troubleshooting session, we found that 90% of one system administrator's time was spent communicating with others, and that misunderstandings greatly delayed finding the solution [1]. In such an environment, there is great potential for improving effectiveness of system administrators by providing tools that allow them to better collaborate, communicate system status, and share control.
2) System administrators frequently create small custom tools to accomplish specific tasks. Administration tasks are complicated given the heterogeneous nature of most systems, with many components from different vendors and distinct local requirements. It is therefore not surprising that many tasks are not directly supported by vendor-supplied tools. We observed system administrators creating small tools on-the-fly to solve problems and collect information. Only one site we observed had a script repository, though the procedures regulating script publishing limited its use.
3) The programming abilities of system administrators vary tremendously. The system administrators we observed were very familiar with shell scripting, but general software and web development skills were usually not part of their job requirements. We observed the majority of scripts created by gurus. Novices would use scripts written by others, but they often did not feel comfortable modifying the scripts or creating their own. Other administrators between the novices and gurus would develop scripts occasionally, but often had trouble understanding and modifying scripts written by others. Administration work could be aided by easier-to-use and reuse scripting environments.
4) Administrators frequently integrate information gathered from different tools and different systems. This is especially true for heterogeneous systems containing components from different vendors in which each component requires a separate tool for access or configuration. Administrating heterogeneous environments could be improved by meta-tools that access and integrate information from many sources.

Successful tools and best practices often evolve and improve when custom tools are easy to create, modify, and share. Although web applications are readily accessible, only advanced users possess the skills to create them. Our goal with A1 is to provide an environment where most system administrators can create, share, and reuse custom web-deployable tools that integrate information from different systems. We want to leverage existing, common scripting abilities so that more administrators can create interactive web apps "for free," without having to become experts in cgi scripting or J2EE servlet programming. We also believe that shareable, web-based tools will be of significant value in aiding system administrators to collaborate.

Design

The primary challenges in creating a scripting language and environment for system administration are ease-of-learning to lower the barrier to entry for busy system administrators, providing broad solutions in heterogeneous environments, and enabling script sharing and reuse for teams of administrators.

To address these challenges, we took a spreadsheet-based approach. Over several decades, spreadsheets have proven to be easy-to-learn tools, particularly because of their straightforward programming style. Many system administrators already use spreadsheets as part of their regular work. The spreadsheet programming model encourages incremental coding, putting part of a calculation in one cell, using the output in a formula in another cell, and so on. Spreadsheets are also more robust than most scripting languages-the nature of spreadsheet execution means that errors only affect dependent cells, and do not necessarily invalidate the entire sheet.

The grid simplifies visual layout significantly, a common problem for less experienced programmers. Finally, the co-location of values and formulas in cells enables users to select a value and see the formula that defined it. This can help users understand and modify others people's sheets for their own use.

A1 extends traditional spreadsheets by (1) permitting cells to contain arbitrary Java objects, in addition to numbers, strings, etc., (2) extending cell formulas to include calls to methods of cell objects, and (3) allowing cells to contain procedural code blocks whose execution is triggered by events in the sheet. As with most changes that increase functionality, these extensions impact ease-of-use, but we believe the impact is small and the benefit large. Our use of Java

takes advantage of a large base of existing libraries, particularly for IT management.

However, it is important to note that in A1 users do not need a deep understanding of Java. It is sufficient to know that objects have methods that can be called to change or return information about the state of the object. Procedural code blocks are triggered in a conceptually straightforward manner, either by changes to the value of specified cells or by cell formulas that evaluate to true. The conceptual model for execution is straightforward: "when something happens, perform these actions." The procedural code itself is written in a simple scripting language, with constructs for assignment, branching, looping, and cell method calls. Users familiar with shell scripting would find the A1 code language similar, and easy-to-learn. We also included traditional spreadsheet help features, such as pop-up menus on cell objects, so that users are not required to memorize all commands, functions, or methods-greatly improving A1's learn-ability through exploration.

A1 can be used to create broad solutions, tying together multiple scripts across different languages and systems. Within an A1 spreadsheet, each cell can invoke a different script and display its output, integrating the output of multiple tools or systems in one spreadsheet. Spreadsheets also bridge the gap between command-line scripts and graphical interfaces. With A1, users can push script output into charts and graphs, easily creating data visualizations of critical system status and performance. Data visualizations are currently under-used simply because they are too hard to create in most command-line environments, and vendor-supplied visualization tools are seldom sufficiently customizable.

However, A1 brings together useful qualities of both command-line (efficiency) and graphical interaction (ease-of-use). The spreadsheet layout also provides ample room for comments and explanation that can be co-located with command output. While all scripting languages permit diligent users to add their comments to help others understand script execution, such comments are often sparse due to programming under time pressure and the fact that comments do not improve script execution or usability; the beneficiary of comments is usually some one other than the script author. Adding comments to a spreadsheet immediately improves the sheet's usability because the comments are visible at execution time and can help clarify the sheet's operation.

To better enable script sharing and reuse, the A1 spreadsheet execution engine supports multiple rendering platforms. Spreadsheets are created using a stand-alone client-based tool with full interactivity, however the same spreadsheets can be saved to a central repository and executed without modification as J2EE-based portlets on a web server. We have integrated A1 with the ISC portal server, which IBM

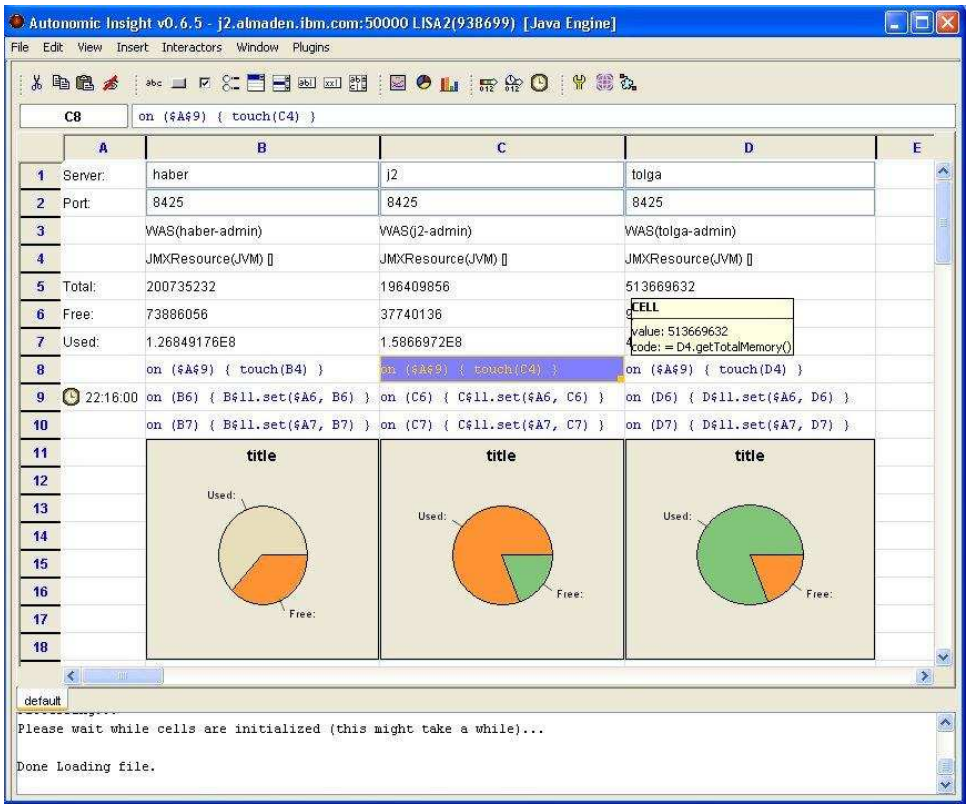

Figure 1: A JVM memory utilization monitor running in the A1 Client.

developed as a one-stop portal for system administration. ISC includes many vendor-provided portlets for managing middleware products. Spreadsheets created by A1 will appear much the same as vendor-supplied system management portlets, with the additional option to edit existing tools or create new tools on-the-fly, extending the toolset with no installation and deployment requirements.

The A1 User Interface

As shown in Figure 1, the A1 interface appears very much like other spreadsheets, with a grid of cells, a cell expression editing field, toolbars and menus. Indeed, A1 supports all standard spreadsheet functionality: cells can contain strings, numbers, dates, and formulas that define a cell's value as a function of other cells. The primary difference is that A1 also supports Java objects as first-class cell contents, object-methods in cell expressions, and event-driven procedural code in cells. A1's enhancements to the spreadsheet language and execution engines enable users to easily build powerful tools that interact with live systems. A1 spreadsheets may also be deployed as web-based portlets. These features are described detail in the following sections.

Objects As First-Class Cell Contents

To support system administrators' needs for control of external systems, A1 adopted an object-based approach. Cells in A1 can contain any Java object, and cell formulas can refer to object methods. For example, if the cell A1 were defined as follows (in this paper, cell definitions will be shown with the cell name in brackets, followed by the contents of the cell):

```
[A1]        java.util.LinkedList()
```

then A1 would hold a Java LinkedList object from the java.util package. Other cells can refer to this object's methods, for example A2 might be defined with a formula such as:

```
[A2]        = A1.size()
```

so that A2 would always display a value indicating A1's linked list's size. References to objects and cell values can be mixed, for example cell A3 might be defined to use the LinkedList get() method, which returns the element at the specified index:

```
[A3]        = A1.get(A2-1)
```

This would cause cell A3 to hold whatever value is at the end of the linked list (the list is zero-based, so the (size-1)th element is the last). Note that A1 uses weak typing when matching objects to parameters. For example, numbers are automatically converted to strings and vice versa depending on the context and methods involved. From a programming standpoint, enabling cells to contain Java objects gives the user access to a large library of objects that provide support for programming needs, such as containers, I/O, statistics, data processing, and networking. From a system administration standpoint, users can immediately leverage a variety of existing administration APIs implemented in Java.

One important issue is how to render Java objects in a spreadsheet. In A1, each object class is associated with an *interactor* that specifies how it will be rendered on the screen and how the user may interact with it. Default interactors have been defined for most common object classes, though A1 allows users to change these defaults. Users can also define interactors for new object classes in their plug-in objects. A1 supports graphical rendering and interaction with objects. For example, a Boolean object can be rendered graphically as a CheckBox, or a NumberCollection can be rendered as an X-Y plot. If there are no interactors defined for an object, it is rendered textually using the default toString() method available in every Java class.

Interactors are designed to work on multiple platforms. In the current implementation, A1 supports three platforms: Java/Swing-based client, HTML-based portal server, and text-based command-line. Each interactor defines how to render and process events in each platform; for example, on the Java/Swing platform the interactor renders a Button and handles button presses in a standard GUI panel, and on the portal server platform the interactor automatically renders the Button as an HTML form input element.

For improved ease-of-use for those unfamiliar with Java, A1 provides a library of Java objects customized for spreadsheet use with a simplified set of methods and predefined interactors. These objects can represent rich data types (e.g., collections, queues, stacks), connections to external systems (e.g., SSH, SNMP, JMX), graphical widgets (e.g., Button, TextBox, ComboBox), and visualizations (e.g., X-Y plot, pie chart). Toolbar buttons exist for creating these objects, allowing the user, for example, to point and click to create an object that manages a server connection. A1 includes a plug-in API for incorporating new Java objects into the framework.

Experienced Java programmers can use this API to enhance existing objects with new interactors or with more sophisticated triggering mechanisms (e.g., pushing events into the spreadsheet, externally triggering spreadsheet re-evaluation). The API also allows customized objects to be added to the toolbar, permitting easier use of new objects in the spreadsheet. We believe the success of A1 will depend to a large degree on a rich set of domain-specific objects permitting access a wide variety of systems. We have provided a core set of spreadsheet friendly objects, but we hope that users take advantage of the API to create and share many more.

Event-driven Procedural and Functional Code

We found the strictly functional programming model of traditional spreadsheets insufficient in defining sophisticated control flows necessary for developing

system administration tools. To enable richer control flow, A1 extends the spreadsheet language to permit cells to include event-driven procedural code blocks. These "micro-scripts" usually include a trigger indicating when the procedural code block should be executed, either by a change in value for any of a list of cells, e.g.,

```
on (<cell address list>)
            { <procedural code block> }
```

or a boolean expression written in the same form as cell formulas, e.g.,

```
when (<boolean expression>)
            { <procedural code block> }
```

Continuing the example above, we might add code to clear the LinkedList contents automatically when its size exceeds 10 elements by putting the following in cell A4:

```
[A4]      when (A1.size() > 10)
                              {A1.clear()}
```

When the object in cell A1 changes, the size method is called. If the size is greater than 10, the linked list is emptied by the clear method. In general, code blocks can be triggered to execute upon changes to cell values, clock ticks, button presses, or any spreadsheet events. Procedural code blocks are written in a simple scripting language containing one or more semicolon delimited statements. A statement can call an object method (e.g., A1.clear()) or assign a new value to a cell (e.g., A10 = A3 * 10).

There is also support for conditional branching using an if statement, and iteration with a for statement (both of which work as in C or Java). When code blocks have no trigger expression, they must be explicitly executed from other code blocks by using the cell address as a statement (e.g., A4()). Code blocks may also force formula re-evaluation and code execution by using the touch statement (e.g., touch A1), which triggers all dependent cells as if the cell value (e.g., A1) has changed.

It is important to note that in A1 users can assign names to cells to improve readability of the code. For example, code in A5 can be named "insert":

```
[A5]      insert: {A1.add("10.0")}
```

and referenced in subsequent code blocks:

```
[A6]      when ( A1.size() < 5)
                              { insert() }
```

For very small tools, using cell references is sufficiently clear. However, as tools grow in complexity, naming cells considerably aids readability and sharability.

Through these event mechanisms, users can achieve rich control flow in their programs. A spreadsheet cell can contain a button that, when pressed, triggers code to connect to a server and perform some action. Regularly polling a remote system can be implemented using a timer object that fires at specified intervals, with polling code triggered by the timer. Any object that uses the A1 plug-in API can also push external events into the spreadsheet, triggering reevaluation of formulas and execution of dependent code.

In addition to for loops, iteration is also supported in A1 through aggregate calls and recursive conditional constructs. For example, to reset a number of servers in cells C1 through C5, one would use the cell range notation ("..") in the method call, e.g., "(A1..C5).reset()". For conditionally bounded iteration, A1's event-based approach provides a simple solution through self triggering when() constructs, e.g.:

```
when (B1 < 10) {B1 = B1 + 1}
```

Web Portal-Based Collaboration Support

To support collaboration among system administrators, A1 spreadsheets can be deployed as portlets in a J2EE-based web portal server. In the web portal, administrators can execute or customize tools deployed by their colleagues from their web browsers. The web portal approach greatly helps shared situational awareness, as different sysadmins can see the same view of system status and controls.

All A1 spreadsheets are initially created in the Java Swing-based client application, which can be launched from within the web portal framework using Java WebStart. For example, Figure 2 shows the tool from Figure 1 running as a portlet. From the client tool, users can either save the spreadsheet locally (for their own use or further development), or to a server repository (for web deployment and sharing with others). Once in the server repository, spreadsheets may be run in a browser, like any other portlet. When a user wants to change a spreadsheet, they can launch the Java Swing-based client tool again using the "Edit" button on the portlet. When the sheet is updated, the user may either save the sheet locally, replace the existing portlet, or save it to the portal under another name. When a sheet is deployed to the portal and running in a web browser, objects are rendered and manipulated using the HTML form-based interactors. In this environment, spreadsheets are rendered to look no different than any other portlet application. To accomplish this, portlet cells that contain code are hidden from the user. Users can also explicitly hide other cells as appropriate.

Implementation Challenges

Implementing A1 involved several challenges, including building the underlying execution engine, and developing rendering and interaction techniques for both Java application and a J2EE-based environments.

The execution engine for a traditional spreadsheet relies on a dependency graph, recomputing each value when there is a change to dependent cells. Side effects are not permitted, and there are no guarantees as to the number of times that a formula might re-execute when triggered. Since A1 models real-world systems, we must handle side effects properly and ensure

that code runs only when necessary. In addition, formulas might include method calls to remote systems that might not return results for an arbitrarily long time. To support these requirements, we implemented a custom multi-threaded execution engine.

One of the biggest challenges was the need to support vastly different interaction models such as the HTML form-based interaction and event-based Java/Swing GUI interaction using a single language and framework. In Swing applications, users can interact with GUI widgets that immediately generate events and cause objects to be rendered to reflect changes. In the HTML form model, input elements are contained in forms that are submitted and processed by a web server.

The difference is that in the web model users can make multiple changes on the form (such as entering text into multiple text fields) and submit the all the changes at once using a button on the form. This minimizes server round trips. A1 handles these two different models uniformly through the use of platform independent interactors and through ‘‘batched event propagation.’’ Batched event propagation registers the current values of all ‘‘delayed’’ input elements (such as text input elements) on a portlet. Upon interaction with an element that causes a server request, all delayed input elements are processed first, before processing the element that initiated the request. Through this approach, A1 accommodates HTML form-based interaction much like the Swing GUI event model.

Sample A1 Tools

Simple System Monitor with SSH

This example demonstrates an A1 SSH object connecting to and executing commands on a remote server (Figure 3). The sample tool includes cells that contain labels and text fields for server name, login name, as well as a password field object used to avoid storing passwords in the script, and to hide them as users are entering passwords:

```
[B1]      "j2.almaden.ibm.com"
[B2]      "eben"
[B3]      PasswordField()
```

The SSH object is created in cell B5, referring to the values from the above cells:

```
[B5]      SSH(B1,B2,B3.getText())
```

Once the connection is successfully established, it can be used to execute commands and assign command output to a spreadsheet cell. Consider the example of a system where occasionally a problem causes the HTTP server processes increase in number. The sysadmin may want to occasionally check to see how many of these processes are running using the ps command. A button labeled ‘‘run’’ is placed in cell A6, with code in B6 that uses the SSH execute() method to run the command whenever the button is pushed, putting the results in cell C6:

```
[A6]      Button("run")
```

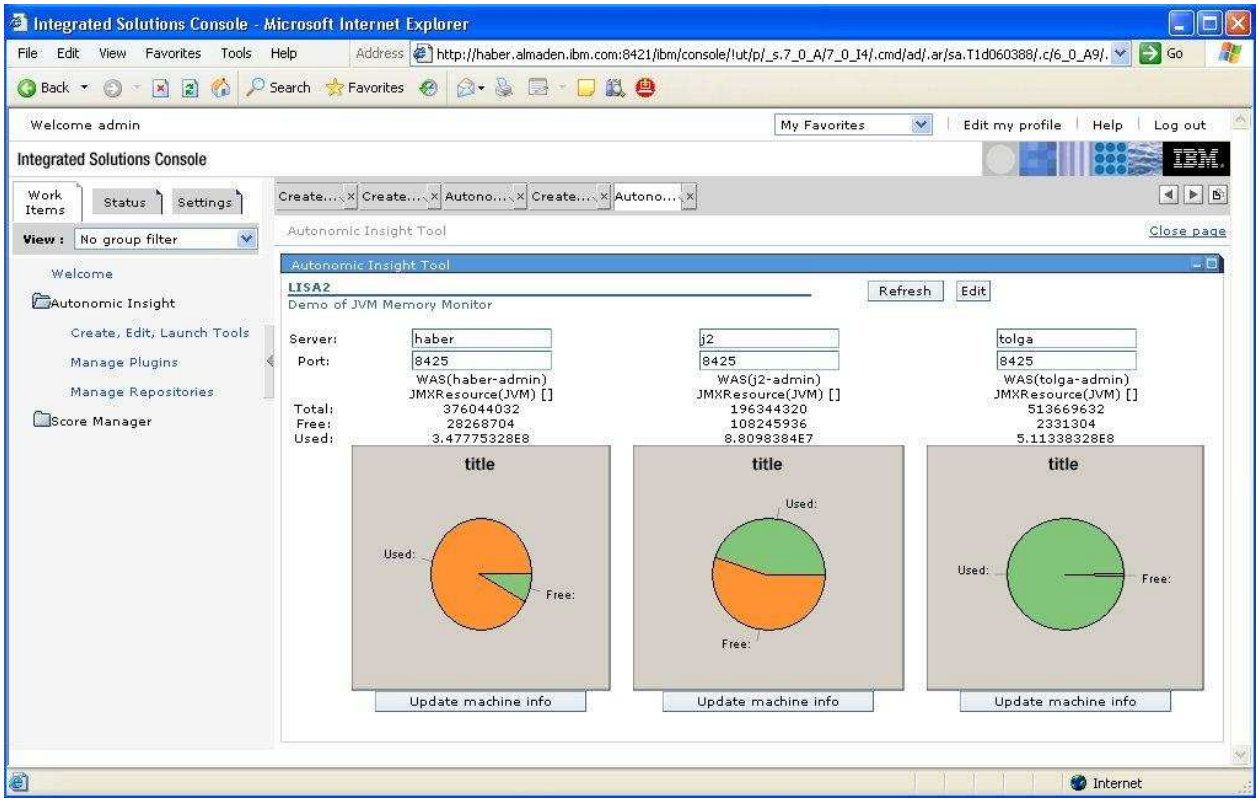

Figure 2: The JVM monitor running as a web portlet.

```
[B6]       on (A6)
         {C6 = B5.execute
                ("ps -ef | grep http")}
```

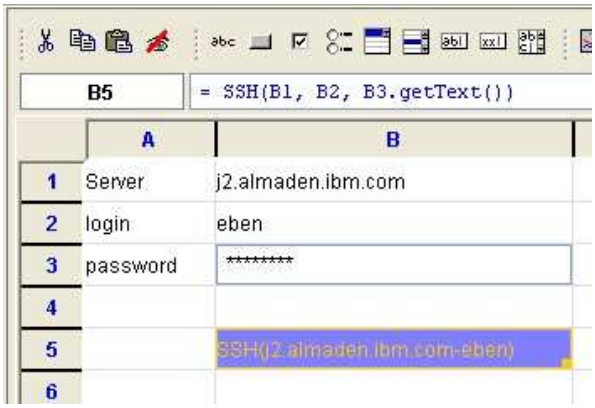

Figure 3: Creating an SSH connection.

The example above can be extended to automatically restart the HTTP server and send email notifications. First, we replace the button with a clock object set to fire every 10 minutes, triggering cell B6 which updates cell C6 with the number of http processes. This, in turn, triggers B7, which checks if this number is greater than 100. If so, it executes a command to restart the http daemon, putting any output in cell C7, and notifies the user by email by manually triggering the code in the cell named "notify" (a.k.a. B8).

```
[A6]       Clock(10*60*1000)
[B7]       when (C6 > 100)
               {C7= B5.execute
          ("/etc/init.d/httpd restart");
                              notify()}
[A8]       MailService
                  ("j2.almaden.ibm.com")
[B8]       notify: {A8.sendMail("root",
                 "eben","http server",
                 "server restarted"+c7)}
```

This example shows how A1 permits the user to quickly create a web-deployable tool that displays and controls the status of a remote system. This tool can be easily shared with other system administrators, who need only change the server, login, and password values to run the script within their own server environment.

Graphical JVM Memory Monitor

Figures 1 and 2 show a tool that uses Java Management Extensions (JMX) to connect to several WebSphere Application Servers (WAS) and display their Java Virtual Machine (JVM) memory usage using a pie chart. In this example, we show how to create that tool in detail.

First, we create labels in column A, and TextField objects in column B into which the user can enter the WebSphere server name and JMX listener port. The object name is visible when the user is typing. Once complete, the object is rendered, in this case as a fully interactive text field (see Figure 6).

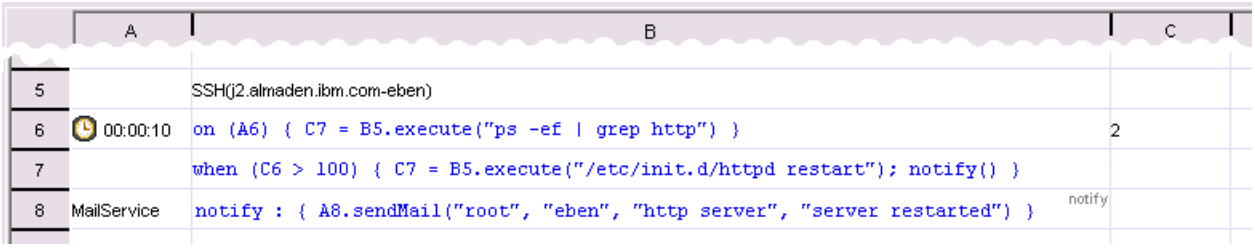

Figure 5: Automating HTTP server restart.

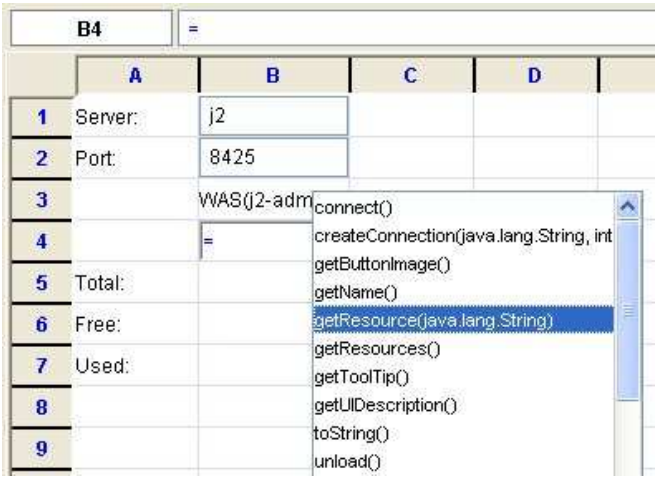

Figure 6: Using a popup menu to list methods.

Next, in cell B3 we create an object that encapsulates a JMX connection to the WebSphere server:

```
[B3]        WAS(B1,B2,"","")
```

The code creating the WAS object refers to the values that the user types in to cells B1 and B2, and uses empty values for login and password (as authentication for JMX is turned off). In JMX, servers can return one or more MBeans, which contain a number of attributes and methods. To retrieve an MBean from a WAS JMX object, we will call the getResource() method in B4, specifically asking for the JVM MBean:

```
[B4]        = B3.getResource("JVM")
```

If users can't remember all methods available on an object, it is possible to right-click on the object to pop-up a method list (Figure 6). Once we have the JVM MBean, we can get further information by using methods such as getTotalMemory() and getFreeMemory() to get the memory statistics into cells B5 and B6, respectively:

```
[B5]        = B4.getTotalMemory()
[B6]        = B4.getFreeMemory()
```

It turns out that the JVM MBean doesn't have a method for returning used memory, but that is easily computed from the difference between total and free using a regular spreadsheet expression in cell B7:

```
[B7]        = B5 - B6
```

Now we have cells that show total, free, and used memory. Because this particular API does not support "pushing" data from the server, we need to regularly poll the server to keep the values up to date. This is accomplished with a timer and code to trigger updates. Cell A9 holds the Clock object,

```
[A9]        Clock()
```

and cell B8 contains code that, whenever the clock fires, "touches" cell B4, causing all dependent cells to be reevaluated:

```
[B8]        on ($A$9) { touch(B4) }
```

Note here that we used the "$" modifier in the cell reference to make it an absolute cell address, thus ensuring that when cell is copied to another location, the expression will still refer to cell A9. This will make it easier to duplicate the code for use with other servers. Now the memory values are being updated at regular intervals. The display can be improved, however, since textual number displays aren't the best way for displaying relative values such as used and free memory. We now create a pie chart object in cell B11:

```
[B11]       PieChart()
```

To populate the pie chart with values, we add code in cell B9 to have a pie slice with a name from A6 ("Free") and value from B6:

```
[B9]        on (B6) { B$11.set($A6, B6) }
```

Thus, whenever the value in B6 changes, the pie slice is updated. In the reference to B$11, only one "$" is used in the cell reference, making the row of the chart invariant. This permits us copy/paste cell B9 to B10, where it will create a pie slice for "Used" memory, and then copy both cells to other columns for other servers. The reference to $A6 makes the column invariant, since the labels are only found in the first column. We now have a complete memory monitor for a single server. Expanding it to support more than one machine is relatively easy. We just select cells B1 through B11, copy and paste them into columns C and D. A1 supports standard spreadsheet cell expression rewriting, so non-absolute cell references in formulas are changed when pasted to refer to the new columns.

The working memory monitor is shown in Figure 1. The memory monitor can also be saved to a server repository where it becomes immediately available as a portlet tool. Figure 2 shows a snapshot of the portlet tool as it appears running in a web browser.

Creating a Server Inventory Report

We have observed system administrators receiving management requests for inventory reports of all servers owned by an organization. The requests came from different people, and involved different groups of servers. The requested information included OS type, version, platform, hardware details (processor type/speed, memory), storage utilization, network interfaces, machine location, etc. The sites involved had various versions of Windows, Unix, Linux, and OS/2, so capturing the required information required a variety of mechanisms, e.g., SSH, SNMP, management tools, and databases. The system administrators we observed collected such information manually, entered into a spreadsheet, created necessary charts, and sent the report to management. Given the nature of this task, it seems ideal to use A1 to both retrieve and organize the data.

Here, we show two examples of such data collection, one in Windows through SNMP and the other in Linux through SSH. Each machine is listed on a separate line, so system administrators can easily duplicate it for as many servers as necessary through copy and paste. From these examples, it should be clear how other server and OS types and access methods could be implemented.

First, we create a set of buttons to trigger updates on system information (Figure 7). In cell A1, we put a button to update all server information, and each line corresponding to a server also has an individual button to specifically update that server's information.

```
[A1]              Button()
[B4,B5,...]  Button()
```

Cells C4, C5, and so on contain code to trigger that row's button whenever the master button in A1 is hit. For example, C4 will contain:

```
[C4]              on ($A$1) {touch(B4)}
```

For each machine, column A will be the machine type/access method for the convenience of the

spreadsheet user copying and pasting rows (e.g., Windows/SNMP, Linux/SSH), and Column D has approach specific code to connect to remote server. For SNMP and SSH the code is:

```
[D4]       on (B4)
               { E4 = SNMP(I4, "", "") }
[D5]       on (B5)
      { E5 = SSH(I5, G5, H5.getText())}
```

Whenever the button is pressed, a connection object is created, causing other values in the row to be updated. Connection status is reported in the fifth column with a simple method call:

```
[F4]       = E4.testConnection()
```

The subsequent three columns include login/password (if needed), and host name. The following columns contain formulas that derive machine information from the connection object. With SNMP, we get the machine description, uptime, #network interfaces, and network interface descriptions as shown below, respectively:

```
= E4.getResourceValue
                    ("1.3.6.1.2.1.1.1.0")
= E4.getResourceValue
                    ("1.3.6.1.2.1.1.3.0")
= E4.getResourceValue
                    ("1.3.6.1.2.1.2.1.0")
= E4.getResourceValues
                ("1.3.6.1.2.1.2.2.1.2.*")
```

The same information can be retrieved from a Linux machine via the following use of the SSH object:

```
= E5.execute("uname -osrv")
= E5.execute("uptime")
= E5.execute("netstat -i | wc -l") - 2
= E5.execute("netstat -i")
```

Other pieces of system information may be extracted in a similar fashion, though some require more extensive textual processing, either using shell commands through SSH, or using one of the built-in string processing functions provided by A1. Though at first it may seem to be complex, once a row collecting information for one machine is completed, it one can be copied and pasted for all other machines of the same type.

Evaluation

A1 is currently an alpha release. We completed one laboratory usability study and are currently conducting field trials. Our laboratory study involved twelve participants, seven professional sysadmins and five programmers. Of the seven system administrators, two had more than four years scripting experience, and the remaining had zero to two years experience. One goal was to determine whether A1 could be learned and successfully used in a two-to-three hour period.

In the study, participants first reviewed a self-paced tutorial of the A1 programming language and user interface. Second, participants practiced using A1 by developing a simple tool following explicit step-by-step instructions. The first two steps typically took about 30 minutes each. Next, participants were asked to develop two system administration tools, one based on the other, given only descriptions of the tool requirements. Finally, an interview was conducted with participants to learn about their experiences.

The first tool to be created was a log-space-monitoring and backup-automation script for an http server. The tasks included querying the http server for the name of its current log disk, using that name to query the file system for log disk utilization, and registering listeners to monitor disk utilization. When the log disk is nearly full, code must notify the system administrator through email, and when the log fills completely, it must stop the http server and start a disk backup. The second tool extended the first one through code to switch log disks automatically when two log disks are available.

After taking the tutorial and building the sample tool, the participants spent between one and two hours working on the subsequent tools. Overall, they were fairly successful, on average completing 80% of the required functionality. Given the complexity of the tools and the limited time to learn the language and environment, these results are very encouraging. Equally encouraging were positive comments by the

	A	B	C	D	E	F	G	H	I
1	Update All								
2									
3	Approach	Update	Code 1	Code 2	Access Object	Status	Login	Password	Host
4	Windows/SNMP	Update	on (A1) { touch(B4) }	on (B4) { E4 = SNMP(I4, "", "") }	SNMP(habert23-)	OK!	<na>	<na>	habert23
5	Linux/SSH	Update	on (A1) { touch(B5) }	on (B5) { E5 = SSH(I5, G5, H5.getText	SSH(j2-eben)	OK!	eben	********	j2

Figure 7: An inventory spreadsheet.

	A	B	C
5		SSH(j2.almaden.ibm.com-eben)	
6	run	on (A6) { C6 = B5.execute("ps -ef | grep http") }	2

Figure 4: Running an SSH command triggered with a button.

participants on the incremental and interpreted nature of A1. One said, ''It allows me to do things in my own order. I can refer to a non-existing object. I know I'm going to create the object next.'' Another said, ''The fact that I can interact with my systems in real time in the spreadsheet that alone is pretty cool.'' Another added that building tools this way was ''the quickest live application [I] ever made.''

The results did highlight several areas for improvement. Learning the object-oriented programming model proved somewhat of a hurdle for the less experienced participants. In addition, many participants had difficulty understanding the semantics of the objects and methods used in the tools. These problems suggest a need for improved introductory documentation to explain the programming model, and better interactive browsing of object APIs to aid programmers in learning object and method semantics.

We are also conducting field trials with system administrators, both inside and outside our company, to see if they can develop tools that are useful in their work settings. So far, A1 has been field-tested in three different groups: web-hosting, server management, and cluster management. The study primarily included requirements gathering, use-case and template development, first-hand experience by system administrators in tool building, and data collection regarding problems and issues with A1 user interface, language, environment, and libraries. In these, A1 was used for developing tools for recycling a steady-state server, creating system inventory reports, and monitoring cluster server performance.

Overall, the reaction to A1 was quite positive. Users especially liked live connections to systems, easy integration of data, and the web interface. One of the users said: ''... but this is like you have real time technical data, and accurate.'' Integration was a major concern among users in their daily work. One user said: ''... this is the only tool that brings all this data together.'' Another referred to the difficulty of creating web interfaces in one of their projects: ''[to get a] web front in front of it (the report) it took six months and it still didn't work right,'' adding ''[with A1] it was on the fly, it is great!'' Users also appreciated the fact that A1 does not require installing agents on servers, pulling data from systems. Issues concerning A1 including lack of support for running scripts as background jobs, need for more interactive scripting and improved text processing utilities, and lack of sufficient documentation and online help on object libraries.

Related Work

Scripting is pervasive in system administration, primarily through command-line shells. System administrators often open a number of shell consoles, execute command-line instructions and automate tasks through languages such as Perl, Python, and C Shell [10]. System management tools are typically vendor-specific-each vendor provides a management tool for its own systems, such as IBM's DB2 Control Center. Tivoli and EMC, two leading providers of tools for system administrators, have hundreds of tools for enterprise-wide operations. To address the fact that system management activities often span multiple tools from multiple vendors, both Tivoli and EMC have integrated environments. For instance, Tivoli's Enterprise Console provides comprehensive management capabilities, including monitoring systems from multiple vendors.

One limitation of these environments is that they provide little support for customization. For example, the systems typically offer standard, prepackaged charting capabilities without a means for end users to create their own system views. PIKT [13] and Cfengine [3] are well known in the system administration community as domain-specific languages for monitoring and configuration management of diverse environments. Unfortunately, complexity of these languages and lack of supporting visual environments make them suitable mainly for experts.

Related work on spreadsheets is extensive in both the research and commercial realms. Spreadsheets were one of the original ''killer apps'' that prompted wide use of personal computers. While effective for tabular calculations, traditional spreadsheets lack expressibility and programming power [17, 4] and they are limited in their ability to interact with external systems. Researchers have taken a variety of approaches to these problems, investigating changes to spreadsheet language, programming model, data types, and user interface.

Of particular relevance is work on object-oriented spreadsheets [2, 5, 9]. These approaches use object models specific to their systems, and require new languages for object definition. Our use of Java has the advantage of a large base of existing users, tools, and libraries, particularly for IT management. In addition, the other approaches were purely functional programming environments, whereas we think our procedural code blocks provide a more natural means of managing the side effects (such as restarting a system) inherent in system administration.

Commercial spreadsheet systems (e.g., Microsoft Excel) address limitations in programming power and access to external systems by permitting users to write programs in an external language (e.g., in Visual Basic). This approach has several drawbacks. First, data sources and processes are not explicitly represented in the spreadsheets as first-class cell contents (unlike numbers, strings, etc.). Thus, interaction with external processes is done through a language and programming model (e.g., Visual Basic) different from that used in the spreadsheet and typically requires different programming skills. Second, functionality

provided by external processes (e.g., operations defined on servers, such as *shutdown*) may not be exposed to the user in a form that can be used readily in spreadsheet expressions. Finally, data are typically pulled from external processes rather than actively pushed by these processes (though in Visual Basic it is possible to program this explicitly too). These issues are all important for system administration.

A1 extends previous work through explicit representation of external systems as objects in cells. These objects expose their properties and methods to the user, who can use them in functional expressions or procedural code blocks to query and control systems. Unlike previous approaches, A1 carefully combines procedural and functional constructs through an event-based approach to achieve rich control structures for the system administration domain. A1 takes the inherently event-based programming of spreadsheets to the extreme, where each cell can either explicitly or implicitly create, listen, combine, and process events. Most importantly, A1's extensions are carefully crafted to create a model and language that conforms to the spreadsheet metaphor while remaining usable, and powerful.

Discussion and Future Work

A1 is a working prototype that aims to support system administrator tool building. There remain open questions as to how well it will work in the real world. We are currently involved in field trials to answer specific questions:

- Is the A1 environment sufficiently easy to use (and reuse) to permit a broader population of system administrators to do their own scripting?
- Is the A1 environment sufficiently powerful and usable to be useful for real system administration tasks?
- Can A1 be learned quickly enough so that administrators are willing to make the time investment to learn it?
- How valuable is a web portal repository of shared tools?
- Does A1 have any scaling limitations for real world tasks?
- What set of access methods (SSH, SNMP, JMX, etc.) is needed?

We hope to better understand how to make A1 successful in the real world through our field trials, and through a public release of A1. There are several areas in which we are already working to improve A1. Error handling and debugging of scripts need improvement, since our extensions to the programming model can make dependencies harder to understand. In addition, right now A1 has no mechanism for logging spreadsheet events, a necessary feature when using A1 to manage systems. We are also actively seeking to develop more plug-ins for connecting to a wider variety of systems through additional protocols.

Conclusion

We have developed a tool-building environment to support system administrators in working collaboratively to create custom solutions for their site. It provides access to a wider audience by combining a spreadsheet-style development environment with web portal deployment. An early version has been made available on the web, and we are actively testing and developing it to make A1 a useful tool for system administrators.

Availability

A1 is scheduled to be available on IBM alphaWorks in October 2005.

Acknowledgments

We thank the system administrators who participated in our field studies and in the preliminary study of A1 for their time and their thoughtful suggestions.

Author Information

Eben Haber works on Human-Computer Interaction at IBM Almaden Research Center. He holds a Ph.D. from the University of Wisconsin-Madison, where he worked on improving user interfaces for database systems. His interests include databases, user interfaces, and the visualization of structured information. He has worked in industry on data mining and visualization, user interface design, and is currently studying human interaction with complex systems in the USER group at IBM Almaden.

Eser Kandogan is a research staff member at IBM Almaden Research Center. He holds a Ph.D. from the University of Maryland, College Park, where he studied computer science with a specialization on Human-Computer Interaction. His current interests include human interaction with complex systems, policy-based system management, ethnographic studies of system administrators, information visualization, and end-user programming.

Allen Cypher is a research staff member at IBM Almaden Research Center. He received a Ph.D. in Computer Science from Yale University. His research interests are in end-user programming, and he has worked in industry designing user interfaces, programming environments and custom tools. Paul Maglio is Senior Manager of Human Systems Research at the IBM Almaden Research Center. In his nine years at IBM Research, Paul has worked and published extensively in the areas of human-computer interaction, intelligent agents, web intermediaries, system management, and autonomic computing. He has a Ph.D. in cognitive science from UCSD, and an SB in computer science and engineering from MIT.

Rob Barrett is a Research Staff Member at the IBM Almaden Research Center in California where he works in the Services Research group on bringing

value from human-computer interaction research to the IBM Global Services organization. His current work focuses on the user experience of system administration and human aspects of autonomic computing. Previous work includes an intermediary approach to designing web applications, optimization of pointing devices, track-following servo systems for tape data storage, and atomic-scale imaging. He holds a Ph.D. in Applied Physics from Stanford University and has earned masters and bachelors degrees in physics, electrical engineering and theology.

References

[1] Barrett, R., E. Kandogan, P. P. Maglio, E. M. Haber, L. A. Takayama, M. Prabaker, "Field Studies of Computer System Administrators: Analysis of System Management Tools and Practices," *Proc. CSCW*, 2004.

[2] Burnett, M., J. Atwood, R. W. Djang, J. Reichwein, H. Gottfried, S. Yang, "Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm," *Journal of Functional Programming*, Vol. 11, Num. 2, pp. 155-206, March 2001.

[3] Burgess, M., "A Site Configuration Engine," *Computing Systems*, Vol. 8, Num. 1, p. 309, MIT Press, Cambridge, MA, Winter, 1995.

[4] Casimir, R., "Real Programmers Don't Use Spreadsheets," *ACM SIGPLAN Notices*, 27, pp. 10-16, June, 1992.

[5] Clack, C., L. Braine, "Object-oriented functional spread-sheets," *Proc. 10th Glasgow Workshop on Functional Programming (GlaFP'97)*, September, 1997.

[6] Couch, A., "An Expectant Chat About Script Maturity," *Proceedings of LISA 2000*, pp. 15-28, 2000.

[7] Gittler, X., K. Beer, "Designing a Configuration Monitoring and Reporting Environment," *Proceedings of LISA '03*, pp. 61-72, 2003.

[8] Hagenmark, B., K. Zadeck, "Site: A Language and System for Configuring Many Computers as One Computer Site," *Proceedings of the Workshop on Large Installation Systems Administration III*, p. 1, USENIX Association, Berkeley, CA, 1989.

[9] Hudson, S., "User Interface Specification Using an Enhanced Spreadsheet Model," *ACM Transactions On Graphics*, 209-239, July, 1994.

[10] Joy, William, *An introduction to C Shell*.

[11] Libes, D., "How to Avoid Learning Expect -or-Automating Interactive Programs," *Proceedings of LISA '96*, 1996.

[12] Myers, B. A., J. F. Pane, A. Ko, "Natural Programming Languages and Environments," *Communications of the ACM*, Vol. 47, pp. 47-52, September, 2004.

[13] Osterlund, R., "PIKT: Problem Informant/Killer Tool," *Proceedings of LISA '00*, pp. 147-165, 2000.

[14] Pierce, C., "The Igor System Administration Tool," *Proceedings of LISA '96*, 1996.

[15] Stepleton, T. "Work-Augmented Laziness with the Los Task Request System," *Proceedings of LISA '02*, pp. 1-12, 2002.

[16] Wack, A., *Partitioning Dependency Graphs for Concurrent Execution: A Parallel Spreadsheet on a Realistically Modeled Message Passing Environment*, Ph.D. Thesis, Department of Computer and Information Sciences, University of Delaware, 1995.

[17] Yoder, A. G., D. L. Cohn, "Real spreadsheets for real programmers," *Proc. ICCL '94*, IEEE Press, pp. 20-30, 1994.