Proceedings of the 7th USENIX Tcl/Tk Conference

Austin, Texas, USA, February 14–18, 2000

SCRIPTICS CONNECT

Eric Melski, Scott Stanton, and John Ousterhout

Scriptics Connect

Eric Melski
Scriptics Corporation
ericm@scriptics.com
Scott Stanton
Scriptics Corporation
stanton@scriptics.com
John Ousterhout
Scriptics Corporation
ouster@scriptics.com

Abstract

Scriptics Connect is a commercial product from Scriptics Corporation that provides an XML-based platform for business-to-business applications. It takes advantage of XML as a standard mechanism for formatting structured data, and uses standard World Wide Web servers and Tcl to provide the infrastructure needed to support business-to-business applications. In addition, it uses Tcl to greatly simplify the task of writing XML-based business-to-business applications.

1 Introduction

XML, the eXtensible Markup Language, has the potential to revolutionize the way that businesses communicate and do business with each other. However, XML alone lacks the infrastructure needed to make it successful. In addition, because of the limitations of the tools available, it is difficult to write XML-based busines-to-business applications. Scriptics Connect was designed to address these issues. Our primary goals when developing Scriptics Connect were:

- Provide the missing infrastructure needed to enable XML-based business-to-business applications. Specifically, provide a means of transporting XML documents and a means of integrating XML-based applications with existing applications.
- Reduce the complexity and difficulty of creating XML-based business-to-business applications

In order to meet these goals, we made use of standard World Wide Web servers to provide a transport mechanism, and we made use of Tcl to provide integration facilities and to simplify XML-based application programming. We chose Tcl as the language of implementation for several reasons. First, Tcl is a natural fit to XML, because it has strong string processing capabilities. Second, the wide variety of extensions available for Tcl make it a good fit for business-to-business applications, which all require some level of integration with existing resources. Finally, Tcl enabled us to develop our application far more rapidly than would have been possible with many other languages.

In this paper, we will begin with a description of XML and why it is interesting for business-to-business applications. Then we will give an overview of Scriptics Connect and the particular problems we wanted to address with the system. We will describe in detail the programming abstractions that we implemented to make XML programming and resource integration easy. Finally, we will discuss problems with our implementation, the lessons we learned from implementing the system, and our future plans for Scriptics Connect.

2 XML

XML, the eXtensible Markup Language, is similar to HTML, the HyperText Markup Language of World Wide Web fame. Both are markup languages; that is, they are used to mark sections of a document with semantic and stylistic information, to allow the document to be better rendered or understood by a machine. HTML, however, is limited because it restricts the types of markup that can be used. It

```
<HTML>
<BODY>
<P>
Ship to:<BR>
XYZ Corporation<BR>
2867 Coast Avenue<BR>
Mountain View, CA 94043<BR>
<TABLE>
    <TR>
        <TH>Artist</TH>
        <TH>CD</TH>
    </TR>
    <TR>
        <TD>Weezer</TD>
        <TD>Pinkerton</TD>
    </TR>
    <TR>
        <TD>The Wolfgang Press</TD>
        <TD>Funky Little Demons</TD>
    </TR>
...
</BODY>
</HTML>
```

Figure 1: HTML encoded CD purchase order

provides one primarily stylistic way of representing your information. It is difficult to represent relationships between parts of your data, or to indicate the meaning of different parts of your data.

By contrast, XML allows the developer to create new markup symbols, called *tags*, to distinguish sections of a document. This means that each developer can create a set of custom tags that map directly to entities in the domain of their work, eliminating ambiguity in the document. This is one of the benefits of XML: it is flexible.

As an example, a purchase order of compact discs in HTML might look like the document in Figure 1. To a human reading this document, the meaning of each section of text is reasonably clear. But a computer reading this document can do little more than render the text according to a predefined set of rules. The computer cannot, for example, easily convert this textual representation into an online database.

However, the same document in XML might look like Figure 2. Now the document is easily readable by a computer. As a side benefit, the document is also easier for people to read. There is no longer any ambiguity about what the document represents, nor what each segment of text within the document means. The relationship between the pieces of data is perfectly clear. This is another important benefit of XML: it provides a clear, easy-to-understand

```
<PurchaseOrder>
    <ShippingAddress>
        <Name>XYZ Corporation</Name>
        <Street>2867 Coast Avenue</Street>
        <City>Mountain View</City>
        <State>CA</State>
        <ZIP>94043</ZIP>
    </ShippingAddress>
    <CD>
        <Artist>Weezer</Artist>
        <Title>Pinkerton</Title>
    </CD>
    <CD>
        <Artist>The Wolfgang Press</Artist>
        <Title>Funky Little Demons</Title>
    </CD>
    ...
</Purchase Order>
```

Figure 2: XML encoded CD purchase order

format for data.

Because XML is an open specification, the software required to read XML documents is readily available from many sources. Several free packages are available for reading the documents, such as `expat` [4]. This means that developers now have an easy way to share data between programs: send it out as an XML document. Developers no longer need complex binary encodings for their data, nor are they dependent on proprietary data sharing systems. This is another benefit of XML: it is an open standard.

Of course, the ability to share data requires that developers agree on a particular set of tags to use for their data. If every developer uses their own set of tags, then the world will be little better off than when HTML was the only practical markup language. Fortunately, standardization efforts are already well underway in many domains. Those efforts will ensure that for many common applications, a set of XML tags already exists.

The primary advantages of XML, therefore, are its flexibility, legibility and its openness. It provides an easy to use and understand format for storing and sharing data.

2.1 Business-to-business applications with XML

Over the next few years, one of XML's uses will be as a data representation for business-to-business applications. In such applications, servers in one company communicate directly with servers in another company to automate business processes. For exam-

ple, the inventory management system of a company might send an electronic purchase order to the sales system of a supplier in order to replenish inventory without any human intervention.

This is not a revolutionary idea; systems already exist that allow the automation of many business processes. However, these systems are based on complex binary encodings of data, such as EDI. The software to decode and process these encodings can cost millions of dollars to deploy within a company. And as with any binary data encoding, it is difficult to extend or customize. XML and the Internet make it easier and less expensive to implement business-to-business applications. They make trading communities possible, in which any company can easily and inexpensively communicate with any other company in the community.

3 Scriptics Connect

For all the potential that XML has to impact the business-to-business world and others, there are some significant problems that must be addressed. The worst of these problems is the sheer difficulty of programming XML applications. Presently, most software available for processing XML consists of low-level XML parsers. Most of these parsers require the programmer to work in a systems programming language like C or Java. As we will show later, using these low-level parsers is hard at best.

A second problem is the lack of infrastructure for XML applications. There is no standard mechanism for transmitting XML documents between entities. And there is no standard for integrating XML applications with other data sources, such as corporate databases or legacy applications.

Because XML is only a data representation, it has no ready solutions for these problems. With Scriptics Connect, our goal is to provide the missing infrastructure needed to make creating XML-based applications simple. The infrastructure components we provide include:

- *transport*: a means for communicating XML documents over the Internet and within an enterprise
- *integration*: a means for moving data between XML documents and other applications

Scriptics Connect combines Tcl and standard Web servers like Apache and Microsoft Internet Information Server with XML to provide this infrastructure. In addition, it provides two levels of programming abstraction to make creating XML applications "falling-over easy."

As shown in Figure 3, a Scriptics Connect installation consists of a World Wide Web server coupled with an XML parsing engine and one or more *document handlers*. A document handler is a Tcl script that describes how to process a particular type of XML document. Document handlers utilize a number of *connection points* to interface with various external applications. The XML parser and connection points are implemented as extensions to Tcl, so Scriptics Connect is essentially the combination of a Web server with an XML-enabled Tcl interpreter.

Scriptics Connect also includes the GUI applications Scriptics Connect Author, which allows the user to easily create document handlers, and Scriptics Connect Debugger, which allows users to debug document handlers installed on their server.

3.1 Scriptics Connect Server

The Scriptics Connect server consists of a Web server coupled with a Tcl interpreter. We chose to support several different popular Web servers in order to make it easier for Scriptics Connect to integrate into a company's existing Web infrastructure.

Building Scriptics Connect on top of a Web server gave us a good answer to one of the infrastructure problems we wanted to address: transport. Using a Web server made it easy to use HTTP as a transport protocol for XML. HTTP is a natural choice for this task for several reasons. First, it is an existing open standard. Second, most corporate firewalls are already configured to pass HTTP requests. One of the biggest problems with existing business-to-business solutions is the use of proprietary transports that require complicated and expensive integration efforts.

The Tcl interpreter in the Scriptics Connect server has been augmented with several extensions. First, it includes extensions for processing and generating XML documents:

- `tclExpat`: a fast, non-validating XML parser based on expat [3, 4]
- `tclDomPro`: an XML Document Object Model interface built on top of tclExpat
- `xmlact`: an API for binding Tcl scripts to parts of an XML document
- `xmlgen`: an API for generating XML documents

Second, it includes several connection points for interfacing with various data sources:

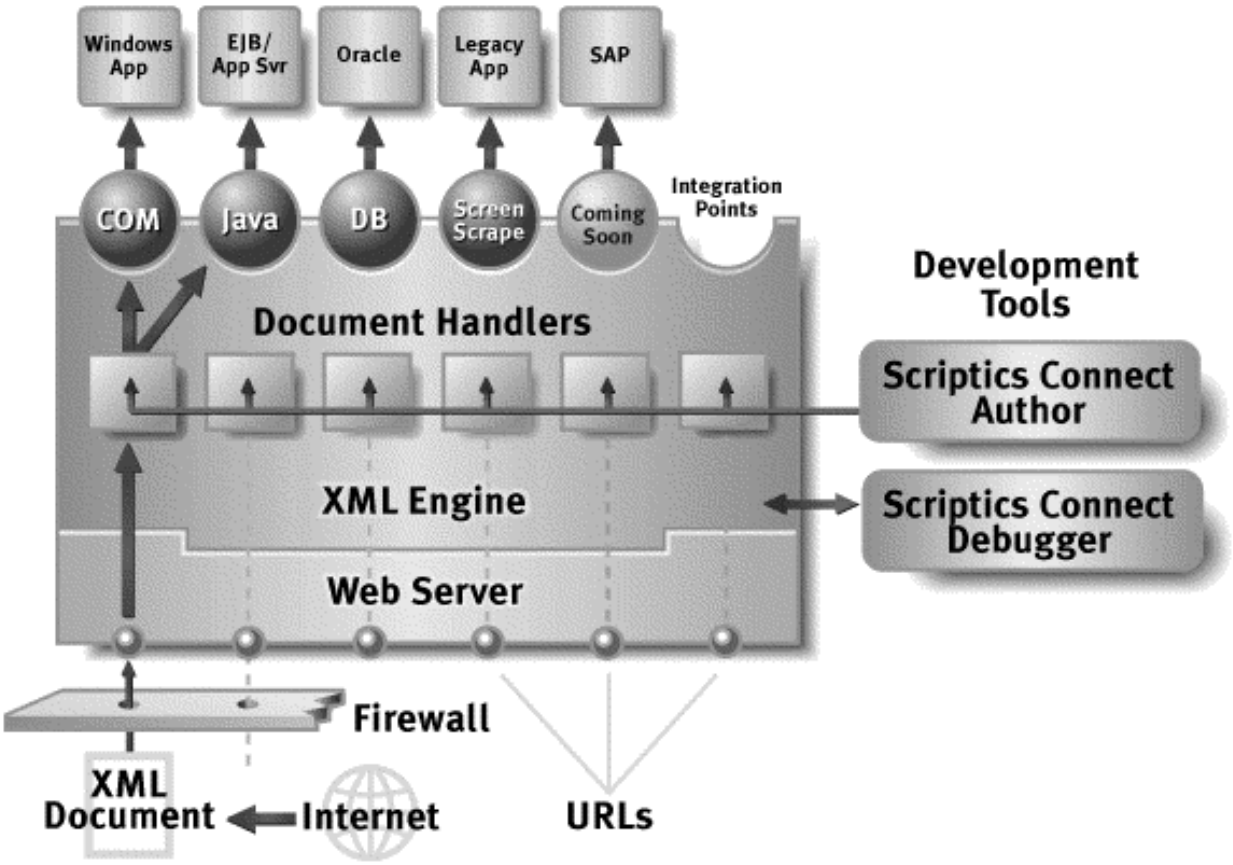

Figure 3: Components in the Scriptics Connect architecture

- `tclCom`: allows COM objects to be created and invoked from Tcl; developed at Scriptics but influenced by the open source tCom extension by Chin Huang [5]
- `TclBlend`: provides access to Java classes, objects, Enterprise JavaBeans and databases, via JDBC [9]
- `OraTcl`: provides access to Oracle databases [8]
- `Expect`: allows Tcl scripts to communicate with and automate applications that normally expect to be interacting with a user typing at a terminal [6]

Tcl is a natural choice for implementing XML applications. Because XML is a text-based format, it is important to have powerful and easy-to-use string and regular expression operations. Tcl's "everything is a string" model is a good fit for manipulating XML data. Additionally, XML is represented using the Unicode character set. As of version 8.1, Tcl uses Unicode as its native character set, which makes manipulating XML data simple. Finally, Tcl has an extensive collection of extensions freely available on the Internet. Many of the connection points in Scriptics Connect come from open source extensions. This allowed us to provide substantially more functionality in our first release than would have been possible if we had implemented everything ourselves.

3.2 XML Document Processing

The combination of a Web server plus the Tcl platform addresses many of the transport and integration issues surrounding XML. However one of the remaining issues we had to deal with was the difficulty of processing XML documents. One of our goals for Scriptics Connect was to raise the level of programming, both to reduce the amount of code that must be written and to reduce the programming skills required to develop business applications.

Most tools for processing XML documents today consist of low-level XML parsers. To use these tools, a programmer must write code in a system programming language like C or Java. These parsers fall into one of two groups: event-based and tree-based. Event-based parsers treat XML documents as streams of data and generate events for each opening and closing tag in the document. Event-based parsers require the user to manage state between callbacks in order to gather data for processing. On the other hand, tree-based parsers treat XML documents as data structures and build in memory a tree representation of an XML document. Using a tree-based parser typically involves writing code that uses Document Object Model (DOM) [1] interfaces to traverse the nodes in the resulting tree to find relevant data. Both models are cumbersome and involve a lot of programming to perform even simple tasks.

As an example, consider the sample XML encoded CD purchase order shown in Figure 2. Suppose a developer wants to import the information in that XML document into a local relational database. Using just the event-based XML parser `expat`, a programmer would have to write C code similar to that in Figure 4. The complexity of extracting the data overwhelms the original programming task of inserting the data into the database. Real XML documents are even more complex than our sample purchase order. This makes XML programming a prime candidate for an abstraction layer.

3.3 The Post-it® Model

Instead of using a strictly tree or event based model, we chose to combine them into a hybrid approach. The metaphor we used to simplify XML programming is that of attaching Post-it® notes to a paper document. Imagine that instead of an XML document, you have a physical copy of the type of document you need to process. If one person were to describe to another person how to process a paper form, they might do it by taking a copy of the form and placing Post-its® on the form, as shown in Figure 5. Each Post-it® would have instructions for processing a particular part of the form, and it might indicate fields from the form that are needed to carry out the instructions. Once a form has been thusly annotated, it could be given to a person who could then carry out the instructions on similar forms as they arrive.

With this abstraction, we can effectively reduce the task of parsing XML to the task of placing Post-it® notes on a document. Pseudo code exploiting this abstraction might look like this:

```
at the CD element call
    addToDatabase with Artist, Title
```

The developer doesn't worry about the technical details of parsing the XML data or finding a position within the document tree. Instead they can focus on the problem they are trying to solve. An important side benefit of our Post-it® abstraction is that it is readily understandable even by non-engineers.

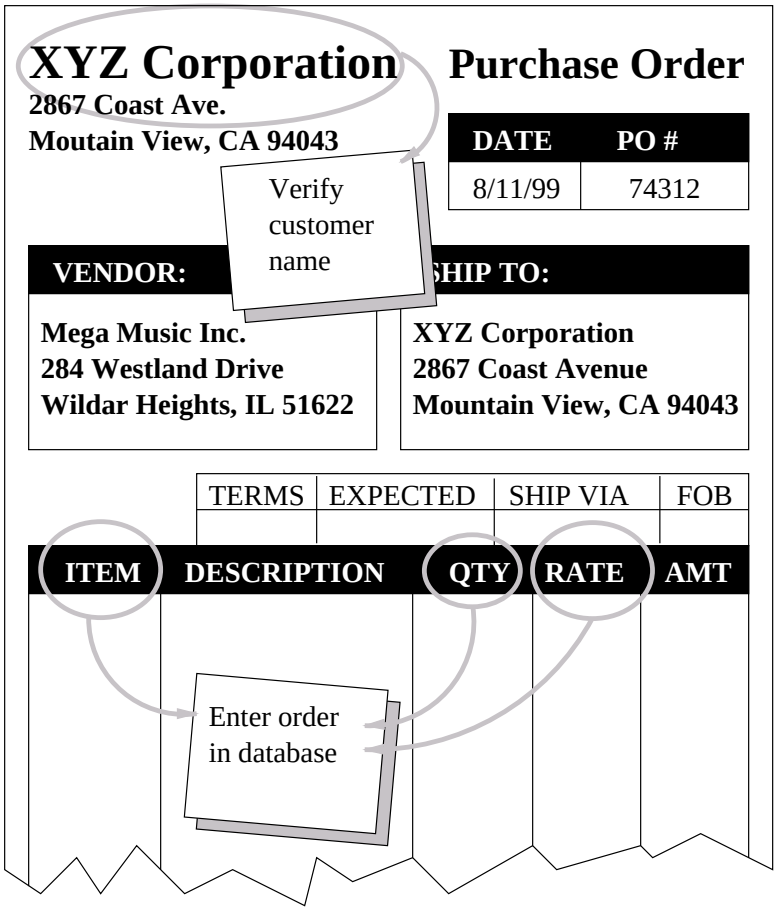

Figure 5: The Post-it® metaphor

```
char currentArtist[128];
char currentTitle[128];
int inArtist, inTitle;
void startElement(void *userData,
                  const char *name,
                  const char **atts)
{
    inArtist =
        (strcmp(name, "Artist") == 0) ? 1 : 0;
    inTitle =
        (strcmp(name, "Title") == 0) ? 1 : 0;
}
void endElement(void *userData,
                const char *name)
{
    if (strcmp(name, "CD") == 0) {
        addToDatabase(currentArtist,
            currentTitle);
        return;
    } else if (strcmp(name, "Artist") == 0) {
        inArtist = 0;
    } else if (strcmp(name, "Title") == 0) {
        inTitle = 0;
    }
    return;
}
void cdataHandler(void *userData,
                  const char *s,
                  int len)
{
    if (inArtist) {
        strncpy(currentArtist, s, len);
        currentArtist[len] = '\0';
    } else if (inTitle) {
        strncpy(currentTitle, s, len);
        currentTitle[len] = '\0';
    }
}
void main() {
    ...
    XML_SetElementHandler(parser,
        startElement, endElement);
    XML_SetCharacterDataHandler(parser,
        cdataHandler);
    ...
}
```

Figure 4: C code for processing an XML encoded CD database

3.4 xmlact: the XML Action API

Our implementation of the Post-it® abstraction took the form of a new API called xmlact. The xmlact API consists of a few commands, `parserCreate`, `parse`, and `parserDelete` for creating, using, and deleting an XML parser. In addition, it has one command, `action`, which is used to associate a Tcl script with a location in an XML document. Locations in an XML document are specified using a path syntax where nested tags are separated by slashes, much like a file name: `PurchaseOrder/CD/Artist`. This is similar to, but simpler and less complete than, the syntax used by XPath [2], a W3C Recommendation for addressing parts of an XML document. These paths are really patterns that are compared with each location in a document to find a match. The parser walks the XML document looking for locations that match a pattern. When a match is found, the corresponding Tcl script is invoked.

Although this basic pattern matching mechanism is a very easy interface that greatly simplifies the task of invoking a Tcl script at a particular place in the document, it doesn't make the task of collecting related pieces of data all that much easier. In order to extract data from the document, the developer would still need to use a mechanism similar to that employed in the earlier C example in Figure 4 to pass data from one action to the next. To simplify the data collection task, we added the ability to refer to data that is contained within the current tag.

Using the xmlact `action` command, the sample C code in Figure 4 can be replaced with the following simple Tcl code:

```
xmlact::action parser CD addToDatabase \
-text Artist -text Title
```

When the XML encoded CD database in Figure 2 is processed, the following sequence of procedure calls will result:

```
addToDatabase Weezer Pinkerton
addToDatabase {The Wolfgang Press} \
{Funky Little Demons}
...
```

A specifier like `-text Artist` indicates that the second word of the Tcl command should consist of the text in the `Artist` subelement of the `CD` element. Several types of specifiers exist, for extracting different parts of the XML document:

- `-text` *tagpath*: retrieve the contents of the specified subelement as text with no embedded tags

- `-value` *tagpath attribute*: retrieve the value of the given XML *attribute* for the specified subelement
- `-attributes` *tagpath*: retrieve a Tcl list of XML attributes and values for the specified subelement
- `-tag`: Retrieve the name of the triggering element
- `-path`: Retrieve the complete path of the triggering element
- `-literal` *string*: Pass the literal *string* through as an argument to the command

These specifiers allow the developer to easily access most pieces of data in an XML document. We deliberately placed a few restrictions on the data available in order to improve performance and reduce the amount of memory required during parsing of large documents. In particular, an action can only refer to data contained within the element it is attached to. Also, we ignore some of the more esoteric XML features like processing instructions. We expect that most business to business applications will not use these features, so we decided to target the "sweet-spot" in order to reduce the complexity of the interface.

3.5 xmlgen: The XML Generation API

There are at least two situations that require some sort of outgoing XML. First, it may be necessary to return data as the result of an incoming request. XML is a natural choice for formatting that response. Second, users of Scriptics Connect may wish to initiate requests as well as respond to them. In order to address these needs, we created the xmlgen API, a means for creating XML on-the-fly.

The xmlgen API has two distinct interfaces for creating XML. To illustrate the difference between the two, consider the following sample XML:

```
<response>
    <header>
        Some text
    </header>
    <body>
        Some more text
    </body>
</response>
```

The first interface is similar to that used in Don Libes `cgi.tcl` [7], in which the document is constructed by a series of nested function calls. The following code uses this interface to create the sample XML shown:

```
xmlgen::element mydoc response {
    xmlgen::element mydoc header {
        xmlgen::text mydoc "Some text"
    }
    xmlgen::element mydoc body {
        xmlgen::text mydoc "Some more text"
    }
}
```

In this style, each call to `xmlgen::element` produces an opening and closing tag with the specified name. The API will execute whatever code is in the body of the call between writing out the opening and closing tags. This style of XML document creation has a couple of benefits: it is easy to see the mapping between the code producing the document and the resulting output, and it works well when the entire document can be produced at once.

However, in some cases, this model does not work well. For example, if you need to incrementally produce the document as you process something, it would be easier to use a more streaming model. In this case, the second interface is useful. This interface uses two commands, `xmlgen::startElement` and `xmlgen::endElement` to create opening and closing tags for elements in the document. The following code uses the streaming interface to create the same sample XML:

```
xmlgen::startElement mydoc response
xmlgen::startElement mydoc header
xmlgen::text mydoc "Some text"
xmlgen::endElement mydoc
xmlgen::startElement mydoc body
xmlgen::text mydoc "Some more text"
xmlgen::endElement mydoc
xmlgen::endElement mydoc
```

One disadvantage of this interface is that it is harder to see the mapping between the code and the resulting document. In addition, the programmer is responsible for maintaining balanced tags, unlike the first interface. However, in some cases, the streaming output model is simply easier to work with.

3.6 Scriptics Connect Author

Although the xmlact interface makes parsing tasks much easier than if users had to write directly to low-level parsing APIs, there is still a lot of coding involved in performing the various integration tasks facing an XML application developer. For example,

it is still troublesome to write the code needed to access a database. To address this issue, we developed Scriptics Connect Author, a GUI tool to assist in creating document handlers.

Author provides two primary benefits. First, it provides a simple graphical interface that helps visualize the task of attaching actions to parts of an XML document. Author displays a schematic view of an XML document; the user selects a particular XML element and associates an action with it. This task effectively generates a call to `xmlact::action`, and is really just a "glossy cover" for the `xmlact::action` function. Figure 6 shows a screenshot of this interface.

Second, through the use of various *wizards*, Author provides easy access to the connection points in Scriptics Connect. Having associated an action with an XML element, the user must define what behavior that action should have. One way to do this is to write a segment of Tcl code. This provides a great deal of flexibility, because the developer can write arbitrary Tcl code. However, we realized that in most cases, the developer does not need that degree of flexibility, and would be better served by an interface customized to a particular type of action. Thus we created a set of action wizards, each of which is optimized for a different type of action, and provides a graphical interface that makes that kind of action easy to implement. For example, we have created a wizard for inserting data into a database; Figure 7 shows a screenshot of this wizard.

We have created several different wizards for Author, including:

- Database wizards for inserting, deleting, updating, and querying
- A condtional wizard for controlling the execution of actions at runtime based on the data in an XML document
- A Tcl variable wizard for extracting data values into Tcl variables for later use
- A TclScript wizard, which allows the developer to supply arbitrary Tcl code, in case our premade wizards are inadequate for their needs

In addition, the Author Wizard API is open, so that users can create new wizards that are tailored to their particular needs.

4 An End-to-End Example

An end-to-end example of how XML can help to automate business-to-business applications begins with two companies, NewCo and OfficeStuff, that do business with each other. NewCo, being a new and rapidly growing company, finds that it often needs to purchase office supplies from OfficeStuff.

Initially, the supply chain proceeds as follows:

1. Employees at NewCo fill out an internal form requesting supplies and send it to the VP of Buying.
2. NewCo's VP of Buying calls her secretary and asks him to order some office supplies.
3. The secretary writes up a purchase order and faxes it to OfficeStuff headquarters.
4. Sales Agent Sal at OfficeStuff receives the purchase order, verifies that NewCo is a real customer and has enough money for the order, then calls the shipping department.
5. An inventory manager at OfficeStuff gathers the items needed to fill the order, updates the inventory database, and arranges for the items to be shipped to NewCo.
6. Sal sends a bill to NewCo.

This system is perfectly functional, but it is time consuming, costly and inefficient. Several people are involved in the chain, each of which increases the cost and chance for error. But the companies could overhaul their supply chain with XML and set up an automatic supply chain, eliminating the need for the extra "middle-men," reducing costs and increasing efficiency and accuracy.

In a partially XML-enabled scenario, NewCo and OfficeStuff, recognizing the inefficiency of their system, get together and agree on the format for the XML documents they will use to automate the supply chain. In this case, there are two: a purchase order and a bill. Now, NewCo can set up an internal purchasing system whereby employees can request office supplies; the orders are collected and formatted as a single XML request, which is then sent via email to Sales Agent Sal at OfficeStuff. Sal, upon receiving the XML document, prints it and processes it much as before. At this stage, half of the supply chain is automated. This is good for NewCo, but OfficeStuff has made no benefit yet. Thus the partially XML-enabled supply chain proceeds as follows:

1. Employees at NewCo log in to the purchase system and request supplies.
2. The purchase system collects requests and sends a purchase order to OfficeStuff.

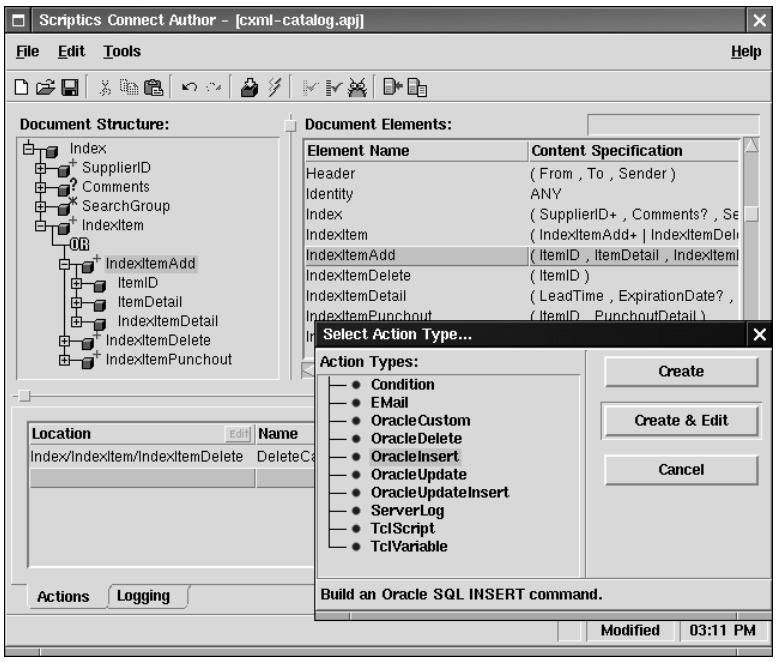

Figure 6: Screenshot of the Author interface

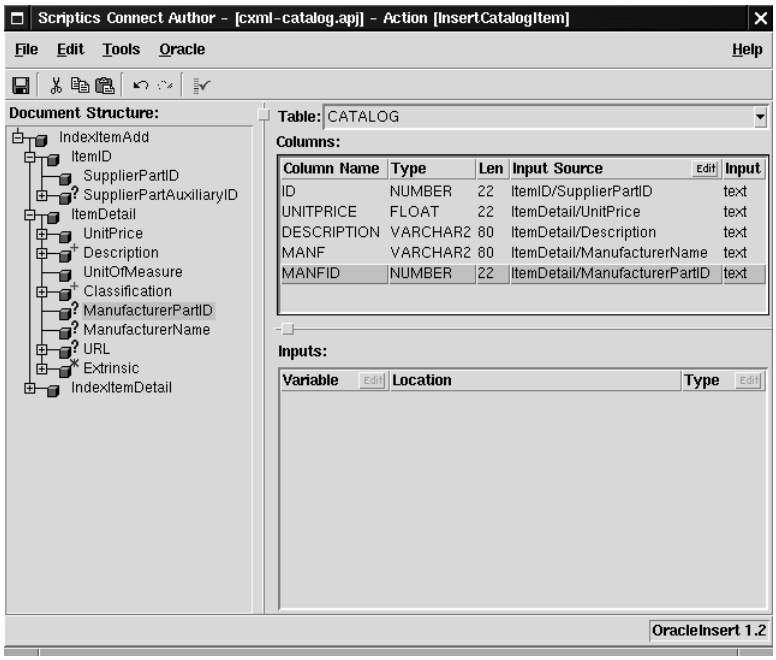

Figure 7: Screenshot of a database insert wizard. The user selects a database table, then drags XML elements and attributes from the tree on the left into the "Columns" listbox on the right to specify which parts of the document to use to fill the database. In addition, the user can drag things to the "Inputs" listbox, to allow simple manipulation of the data before inserting it into the database.

3. Sales Agent Sal at OfficeStuff receives the purchase order, verifies that NewCo is a real customer and has enough money for the order, then calls the shipping department.

4. An inventory manager at OfficeStuff gathers the items needed to fill the order, updates the inventory database, and arranges for the items to be shipped to NewCo.

5. Sal sends a bill to NewCo.

With a system like Scriptics Connect, OfficeStuff can automate their half of the supply chain as well. They can create a document handler to process the XML-based purchase orders from NewCo. This document handler might connect to the OfficeStuff customer database to verify the validity of the customer and the depth of the customer's pocketbook. It could connect to the inventory database, verify that sufficient inventory was available, update the inventory, and issue an order to the inventory manager to load the furniture on the delivery truck. It could even issue an appropriate bill to NewCo for the purchase.

After creating the document handler, OfficeStuff can publish it to their Scriptics Connect server and notify NewCo of the URL to use when sending purchase orders. After NewCo has adjusted their internal purchase system to use HTTP instead of email for delivering the purchase order, the supply chain will be as automated as it can be, unless some sort of machine is used to load the delivery truck. The system now proceeds as follows:

1. Employees at NewCo log in to the purchase system and request supplies.

2. The purchase system collects requests and sends a purchase order to OfficeStuff.

3. OfficeStuff's Scriptics Connect server verifies the validity of the customer, checks the customer's credit line, checks and updates the inventory database, sends a shipping request to the shipping department and bills NewCo.

4. An inventory manager at OfficeStuff gathers the items needed to fill the order and arranges for the items to be shipped to NewCo.

Instead of the several people involved in the original purchase, only the bare minimum people required are involved. The savings to both companies in this case is probably not tremendous, but it probably is noticeable. And OfficeStuff can now turn to its other customers and ask them to use XML as well, further reducing their operating costs. Similarly, NewCo can ask its other suppliers to use XML-based systems. As more and more entities become XML-enabled, those companies that are automated will enjoy further reduced costs.

5 Problems with Scriptics Connect

Although we feel we have been successful in achieving our initial goals, we recognize that there are some problems and shortcomings with our implementation.

One obvious shortcoming is in the area of generating XML. Our system works very well for receiving XML, but it is difficult to generate XML for transmission. The xmlgen API is completely functional, but it is not very user-friendly. We would like to provide graphical facilities for describing how to generate XML. Ideally, such a GUI would provide the same simplicity that Author provides for handling incoming documents.

Another area that needs improvement is data collection. Our current facilities for gathering data from an XML document are complete, but they are difficult to apply to some situations. For example, retrieving the data from all of the instances of a repeated element in an XML document is not straightforward. The user must create a TclScript action and append values to a list. Complex data structures are another example. Presently, the user can extract the data, but must do so one field at a time, which can be tedious. In both cases, we believe that we can create new mechanisms to address the problem.

A third area is code and action reuse. It is currently not possible to share actions between document handlers. It is easy to imagine scenarios that would call for the sharing of actions, making this a potentially large inconvenience for our users.

6 Future Work

We have an aggressive development schedule set for Scriptics Connect. In version 2.0, we plan to add several features:

- Built-in integration with online trading communities like AribaNet and CommerceOne.

- Additional transport mechanisms for receiving and sending XML, such as FTP and SMTP

- Better monitoring tools

In version 3.0, we plan to further enhance the GUI, perhaps integrating Author into an IDE for working with Scriptics Connect.

7 Conclusions

XML has the potential to greatly reduce operating costs and increase efficiency and accuracy in many business-to-business applications. However, the lack of infrastructure and the difficulty of creating XML applications are two significant barriers to the spread of XML. We had two goals in mind when creating Scriptics Connect:

- Provide the missing infrastructure needed to enable XML-based business-to-business applications
- Simplify the process of creating XML-based business-to-business applications

We achieved the first goal through the use of standard World Wide Web servers, to provide a transport mechanism, and Tcl, to provide an integration mechanism. We achieved the second goal with the xmlact API and the Scriptics Connect Author. Tcl proved to be an ideal choice for XML-based business-to-business applications, because of its ability to handle XML well, and because of its ability to connect to many external applications.

8 Acknowledgements

We would like to thank the employees of Scriptics Corporation, without whom Scriptics Connect could not have been created. In addition, we would like to thank the authors of the open-source software packages that we made use of in Scriptics Connect.

Post-it® is a registered trademark of 3M.

References

[1] Document Object Model: http://www.w3c.org/DOM. World Wide Web Consortium.

[2] XPath: http://www.w3c.org/TR/xpath.html. World Wide Web Consortium.

[3] Steve Ball. XML Support for Tcl. In *Proceedings of the 6th Annual Tcl/Tk Workshop*, page 109. USENIX, September 1998.

[4] James Clark. Expat Home Page: http://www.jclark.com/xml/expat.html.

[5] Chin Huang. tCOM: http://www.vex.net/~cthuang/tcom/.

[6] Don Libes. *Exploring Expect.* O'Reilly & Associates, Inc., 1994.

[7] Don Libes. Writing CGI Scriptics in Tcl. In *Proceedings of the 4th Annual Tcl/Tk Workshop '96*, pages 189–201. USENIX, July 1996.

[8] Tom Poindexter. OraTcl: http://www.nyx.net/~tpoindex/tcl.html.

[9] Scott Stanton. TclBlend: Blending Tcl and Java. *Dr. Dobb's Journal*, February 1998.