Lightweight Integration for Scripted Hypermedia*

Vance Maverick† and Michael A. Harrison
Computer Science Division – EECS
University of California, Berkeley

April 10, 1995
Submitted to ACM Multimedia '95

Abstract

It is widely agreed today that hypermedia systems are most useful when they are open, that is, can cooperate with other applications. Research in this area has concentrated on the problems of handling data and links in the multiple-application context. But there is an important class of *scripted* hypermedia documents, such as 'guided tours', which specify not only the linked data, but also complex instructions for its presentation and its interaction with the user. Supporting such documents in an open environment requires a protocol for the dynamic management of objects, general enough to handle complex user-interface objects and data display states.

In this paper we present Jicama, a simple integration framework based on Tcl/Tk [21]. A compliant application supplies *objects*, corresponding to top-level Tk windows. Each object belongs to a type, which is implemented by a Tcl procedure in its application, with a standardized interface. A central server maintains a registry of all objects and the relationships between them. Objects are registered by handles; all the details of implementation and storage are up to the individual application.

We have implemented Jicama, and used it to construct a variety of hypermedia documents. We have created six simple object types that conform to the model, and two interactive applications: a hypertext, and a 'guided tour' scripting tool which models user interactions as a graph of interaction states. Adding Jicama support to existing programs is easy: we ported a substantial application to the model by adding only about 100 lines of new code. Finally, Jicama is equal to the dynamic needs of many hypermedia models, and makes them fully interoperable.

1 Introduction

There is widespread interest today in complex, highly interconnected multimedia documents. Hypertext and hypermedia systems have been developing for years [8, 25], and recently the World-Wide Web [2], a simple distributed hypermedia system, has become enormously popular.

In these systems, the user is usually free to traverse any available links. But there is also a need for a more programmed browsing style, in which the author can guide the user along a particular path through the information. Examples of such systems are Zellweger's scripted documents [26] or Trigg's guided tours [23]. For this paper, we refer to these generically as *scripted documents*, whatever the model.

It has long been recognized [7] that hypertext and hypermedia systems are most valuable when they are open, that is, can easily be extended to handle new data types, or can even interact directly with third-party applications. Such systems as Microcosm [9] and Chimera [1] approach this problem by defining a universal data model, and bringing applications into varying degrees of compatibility with it.

*This paper is based on work supported by the National Science Foundation under Infrastructure Grants Nos. CDA-8722788 and CDA-9401156, and by the Advanced Research Projects Agency (ARPA) under Grant No. MDA972-92-J-1028. The content of this paper does not necessarily reflect the position or the policy of the Government.

†For correspondence: `maverick@cs.berkeley.edu` or Vance Maverick, Computer Science Division, 571 Soda Hall # 1776, Berkeley, CA 94720-1776

Scripted hypermedia documents, though, consist of more than just linked data. They also include complex instructions for its presentation to the user, including display specifications and user interface elements. Existing hypermedia systems do include some user-interface support: Trellis [10], for example, allows the author of a script to create labelled buttons which appear in parallel with data items. HTML [3] provides an elaborate sublanguage for the definition of 'forms', which allow the user to communicate complex state information to the server. But for an open hypermedia system to support scripted documents, a dynamic interaction protocol is required which will enable applications to cooperate in creating, displaying and destroying on-screen objects, whether they are user-interface objects, data objects, or something in between.

The Tcl/Tk user-interface framework [21] is a promising foundation for such a protocol. It provides a straightforward scripting language which is easy to integrate with C code; an adaptation of the X Toolkit 'widget' [16] to the language; and a clean, useful set of readymade widgets. Numerous developers have used it to create widgets and small applications, including display tools for many common data types.

When Tk was first presented [20], special emphasis was placed on its `send` command, an interprocess communication facility. It was predicted that this would "[make] it possible to replace monolithic applications with collections of reusable tools." But this prediction has not yet been fulfilled. The widgets of Tk and Xt made possible the integration of user-interface components which was promised by the X window hierarchy; but the low-level `send` command is still waiting for a high-level framework which will make it easy for one application to call on another.

In this paper, we introduce a new integration framework, Jicama[1], to fill this gap. In Jicama, applications are responsible for the management of *objects*, corresponding to top-level windows on the screen. Each object belongs to a type, which is implemented by a Tcl procedure in its application. This procedure must follow a standard interface. A central registry keeps track of all the objects. An object asserts its requirement of another object by presenting a specification to the registry; when this requirement has been satisfied, it is maintained until explicitly released.

This registry does not access objects directly; instead, it manages them through handles. The application which supplies an object is free to choose an appropriate implementation, storage, and run-time representation. (By contrast, a 'heavyweight' integration framework such as Microsoft's OLE [18] imposes complex standards for all three design choices.) This makes it easier to port applications to Jicama, and also to coordinate separate applications. The registry works the same whether an object's implementation is in the same or a separate process, or even on a remote machine.

In practice, Jicama is a flexible system for designing multimedia interactions. We have demonstrated its power by building six elementary object types and two sample hypermedia models: hypertext, and a guided tour modeled as an automaton. These models are themselves Jicama object types, so one may easily call up another. Adapting existing applications to the model turns out to be easy, requiring no more than about 100 lines of new code.

In Section 2, we present the model and its Tcl implementation. Section 3 shows how building hypermedia works in Jicama, from elementary objects to the two hypermedia models. In Section 4, we report our experience in bringing a pre-existing application into conformity with the system. Finally, we discuss the future of Jicama (Section 5), and summarize our conclusions (Section 6).

2 The Jicama Programming Model

2.1 Objects and requirements

Under Jicama, the dynamic behavior of a scripted document is modeled as a set of discrete objects, which come and go at discrete times. The general mechanism governing their creation and deletion is the *requirement*: one object asserts that it requires another. This relationship endures until one of the objects is deleted, or the first object explicitly releases the requirement.

One object may satisfy several requirements; for example, if a hypertext document includes many links to a single file of source code text, the user need not see a separate copy of the source file for every link which is currently being followed. At any given moment, then, the existing objects are the nodes of a directed

[1] Jicama is a crisp vegetable often found in salads, like daikon in appearance and texture but very mild.

graph, and the requirements are the edges connecting them. When an object is no longer required, a simple garbage collection causes it to be deleted.

Let us examine more closely what happens when the user follows this hypertext link. In Jicama terms, the top-level menu or shell of the application is an object; when the user picks a file, this object asserts its requirement of the hypertext document. When the user follows the link, the hypertext object requires the source code file. This sequence is shown in Figure 1, in panels *a* and *b*.

2.2 Implementation of the model

In Jicama, the graph of objects and requirements is not maintained as explicit pointers between data structures, in the manner of OLE [18]. In fact, the storage, run-time representation, and implementation of an object are entirely up to the application which supplies it. The object must, however, be identifiable by a unique handle. A central registry stores these handles and maintains the graph relationship between them.

Every object belongs to a type. This type is implemented as a Tcl procedure in the object's application. (From now on, we will use the standard Tcl shorthand 'proc' for 'procedure'.) Jicama defines the interface for each object type proc, and for the central object registry. Figure 2 shows this arrangement schematically.

Each Jicama application is linked with a standard library, and calls a standard initialization function on startup. This function checks whether a registry server already exists; if not, it creates one. Within the 'primary' process, containing an application and the server, calls to the object type and registry interfaces can be made by direct Tcl calls. With a remote process, these calls must be implemented by Tcl `send`. The difference is transparent to the programmer.

2.3 Processing Requirements

When one object needs to bring another up, it asserts a requirement by calling a registry interface function, passing a *specification* of the object it needs. (The complete registry interface is given in Appendix A.1.) This specification consists of two parts: the object type, and an 'argument list' containing whatever data, filenames, display parameters, etc., are appropriate for the type.

To satisfy this requirement, the registry first compiles a list of all the current objects of the specified type, and the arguments from their specifications. It then calls on the Tcl proc corresponding to the type, in whatever application, and asks it to satisfy the requirement. (The full interface of an object type proc is given in Appendix A.2.)

The object type proc compares the new specification to the specifications of the existing objects. If one of these matches, the proc simply returns it; if not, the proc creates and returns a new object, matching the new requirement. Deciding whether two specifications match is entirely up to the object type proc; Jicama itself never examines the arguments of a specification at all.

The registry saves the object handle returned from this call, and records a requirement, directed from the calling object to the object which matched the specification. A handle to this requirement is returned to the caller.

Some objects in the graph are marked as permanent. (In our main sample application, the only permanent object is the top-level user menu.) Any object which is not required, directly or indirectly, by a permanent object, is garbage. In an early implementation of Jicama, we used reference counting to identify garbage. Currently, we use mark and sweep, for several reasons. It correctly cleans up graphs with cyclical regions, which are a real possibility in closely linked hypertext. Also, it allows the cleanup, which can trigger a long and visually drastic sequence of window deletions, to be deferred until an appropriate time. Garbage collection is only performed on request.[2]

Figure 1 shows how this process works. The requirements asserted in panels *b* and *c* specify the same file, but different lines to be highlighted. In this case, the object type proc decides that both requirements can be satisfied by the same object. If the highlight regions had been far apart, the type proc could have chosen

[2] OLE uses reference counting to manage 'component' objects. Cycles are a problem, as the documentation warns, but this choice makes sense; under OLE, complete documents composed of hundreds of objects may be in memory at once, so mark and sweep would be unacceptably slow.

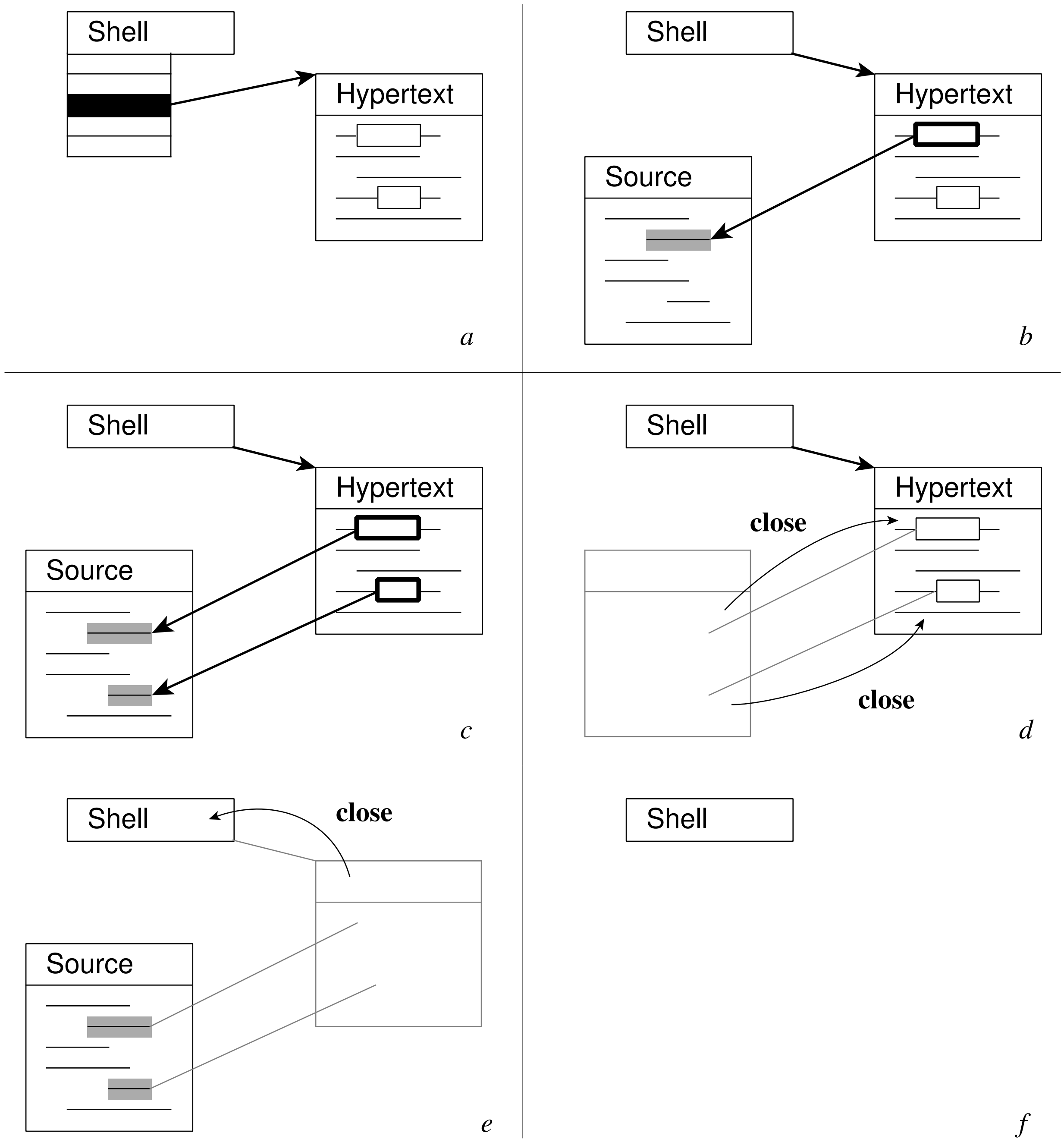

Figure 1: Requirements, events, and garbage collection on the graph of objects. *a:* The user opens a hypertext document from a menu shell. *b:* The user follows a link from the hypertext document to a source code file. (Figure 3 shows the actual screen appearance at this point.) *c:* The user follows a second link. Because it points to a line in the same file, the requirement can be satisfied by the same object. *d:* The user closes the source code file. The `close` event is propagated along both requirements, alerting the hypertext file to change its display state. *e:* Alternatively, the user closes the hypertext file. All requirements are outdated, and the `close` event is propagated back to the shell. Now, *f:* in response to the `close` event, the shell triggers a garbage collection. The shell remains, because it is marked 'permanent'; the source code file is closed, because it is no longer required.

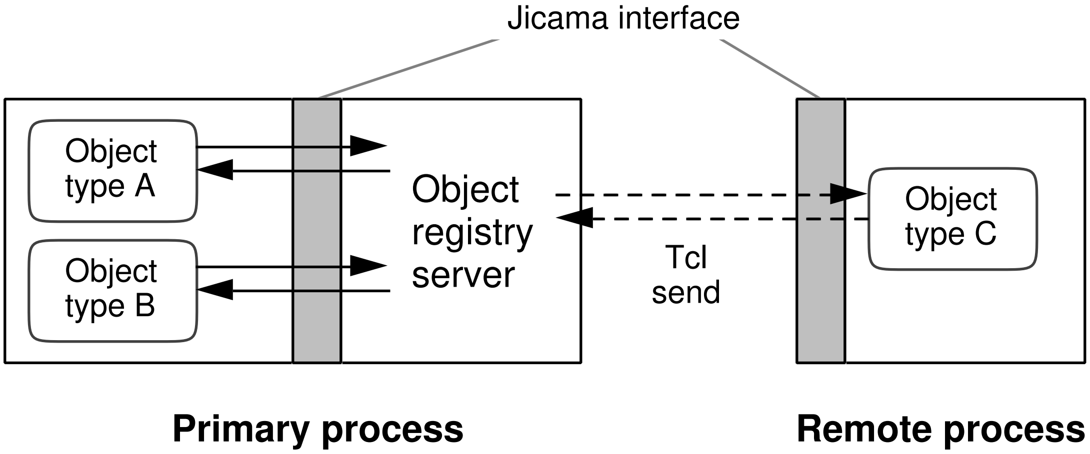

Figure 2: Communication between processes in the Jicama system. Each process implements some number of object types. These communicate with a central object registry server through the Jicama interface, possibly implemented by Tcl `send`. The registry only holds handles to the objects: storage and implementation details are up to the application.

to create a second source-code object to satisfy the requirement, or perhaps to split the object's window, and show one highlight region in each panel. The decision is up to the implementor of the object type.

2.4 Events

The Jicama model also supports communication between objects, through an 'event' mechanism which takes two forms. An object posts an event by calling a central system service. If it is an ordinary event, all objects which require it are notified. If it is a broadcast event, Jicama notifies all objects which have registered themselves as broadcast clients. (These events are similar to the communication mechanisms in other integration frameworks, such as Field [22].)

Ordinary events enable objects to learn of changes in the state of the objects they have required. For example, our hypertext viewer makes the appearance of its links reflect whether their targets are currently visible. When the user follows a link, it changes as the target object is brought up. When the target object is closed, the viewer catches the close event and restores the appearance of the link. Broadcast events are discussed further in Section 3.3.2.

2.5 Special object services

As we have said, the arguments in a Jicama object specification are defined by the application which implements the object. However, two needs are common enough that we support them with special services: automatic object placement and file collection management. In each case, a Jicama utility function scans the argument list for conventional arguments.

The Jicama object placement service can place objects precisely at a position given in X and Y coordinates (expressed as fractions of the screen width and height), but in practice, approximate placement is more useful. The author of a script specifies a position in terms of a rough 'quadrant' of the screen (center, northeast, south, etc.); the placement service searches that quadrant to find the position which will make the object overlap the smallest area of other material already there. A standard argument (`-autoplace` {*quadrant*}) is supported by all object types; they pass it on to the placement service when they create a new object.

Hypermedia documents may consist of many distinct files. (Most objects in the Jicama prototype read a whole file for their content, so a complex document consists of one file per object, or more.) This leads to the usual portability problem: how can the author specify filenames in links so the links may be followed regardless of the reader's 'current directory', but the document is still valid if moved to another directory or machine? Jicama provides a very simple 'file collection' mechanism that is used like a directory in resolving filenames. On disk, it contains a list of filenames; when it is read, the names are all resolved, taken as relative to the directory containing the collection. Thereafter, the collection is used as a mapping from short filenames to full paths. The collection is not loaded again, but simply referenced by its own short name; Jicama provides a utility to scan argument lists for the elements `-file` *file-name* and `-collection` *coll-name*, returning a full path.

This scheme introduces a consistency-maintenance problem: each link in a complex hypermedia document must name the right collection. To alleviate this problem, we made a simple change in our scripting objects; if they are invoked with a collection argument, they pass it on to any objects they require — unless, of course, the author has named another collection explicitly in that link. Thus, most links in the stored document do not specify a collection directly at all.

3 Scripting under Jicama

3.1 Simple object types

As a basis for complex scripted documents, we have implemented four fundamental object types: text, graphics, sound and video. None of these is rich enough to be used much on its own; rather, they are called on by scripts created with more complex tools.

The TextDoc object, based on the Tk text widget, is the most sophisticated. In addition to displaying a text file, it can highlight ranges; it can satisfy any number of requirements at once, as long as their highlight ranges fall in the single visible region.

The Sound and Video objects support only crude synchronization. When they come up, they can be made to start play right away, or to wait till the user presses a button. When they stop play, by reaching the natural end of the data, they post a `stop` event; the object which requires them can use this to trigger more requirements.

There is also one user-interface object type, a dialog box. It displays a message and a row of labeled buttons, all specified by its argument list. When the user clicks on a button, an event is posted with the corresponding name.

Finally, there is a menu shell, from which the user may open any of the fundamental object types, or either of the more complex types we describe below.

3.2 Hypertext

We have implemented an example of hypertext in Jicama, with the FancyText object. Like the TextDoc object, its implementation is based on the Tcl text widget, but it interprets its input file as tagged structured text, in the manner of SGML and HTML [12, 3]. It also reads a secondary input file containing a set of formatting instructions, associating font, size, color, etc. with the tags. The tags also can function as the anchors of links, either the source or the target.

Links are represented as object specifications. For example, in Figure 3, we show a link from a word in a file of software documentation to a location in the source code illustrating the text. Just as an HTML document specifies a link with the HREF attribute of its anchor tag, the link tag in this document uses the REQUIRE attribute:

```
<LINK REQUIRE={
{TextDoc {-file Sound.tcl -highlight 49.0 49.21 -autoplace sw}}
}>here</LINK>
```

When the FancyText object is closed, the code file will be released on the next sweep — unless it is required by another document, for instance another documentation file with a link to the same code, or a 'project' within the software development environment, through which the user is editing the file.

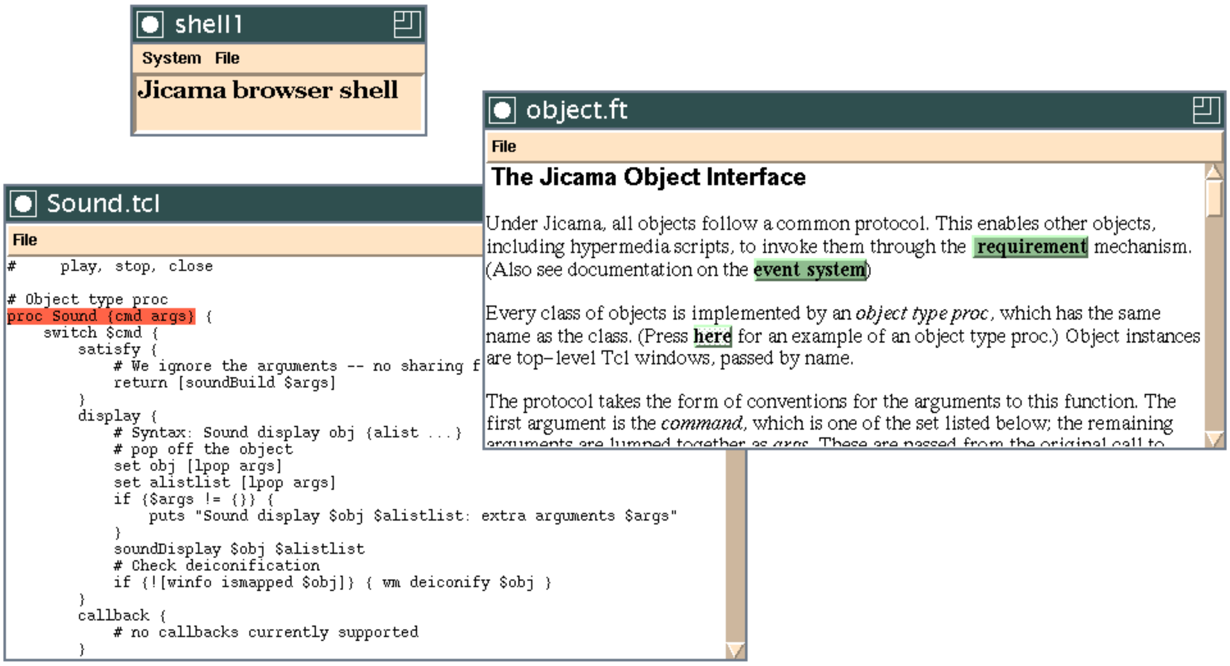

Figure 3: Hypertext using Jicama. From the top-level 'shell' object, the user has opened a page of software documentation, and followed a link ('here') to an illustration in the code.

3.3 DFA interaction model

The most complex scripting system developed under Jicama is based on a model of user interaction as a deterministic finite-state automaton (DFA). With it, the user may construct 'guided tours'. While the interaction lasts, the system is considered as a set of states; it is always in exactly one. It begins in a designated state (named 'start', by convention), then changes state in response to user actions, following predefined transitions.

The states affect what the user sees by means of the requirement mechanism; each one contains a list of object specifications. As the DFA enters a state, it asserts the requirements specified there, and releases any requirements from the previous state; then it triggers a garbage collection to clean up. (If an object is required in both states, it will remain on the screen, without being deleted and redrawn, because mark-sweep allows collection to be deferred.)

Figure 4 shows the graph structure of a typical DFA-based document. It defines a simple 'guided tour' of the Jicama system itself. The tour begins with an introductory sequence of two nodes; from the node 'main', though, the user may visit the three nodes 'text', 'dialog', and 'sound' any number of times, before proceeding to the last one, 'script'.

The DFA object, which implements this model, allows simple point-and-click editing of these graphs. Nodes may be rearranged freely for legibility. As builders of other graph-based systems [15] have found, highly connected graphs quickly become unreadable beyond a fairly modest size. They may be made more readable by breaking them up into subgraphs. (While this helps clean up each individual DFA, it makes it difficult for the author to oversee the whole. We discuss this further in Section 5.)

Since a DFA is itself just another Jicama object, it may be invoked at runtime by another DFA. To enable this form of procedural abstraction, DFAs support a standard specification argument, `-play`. When brought up in 'play mode', a DFA immediately begins executing at its start node.

3.3.1 Editing object specifications

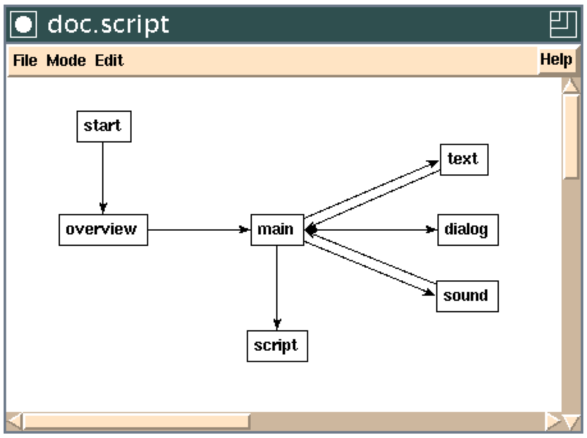

Figure 4: Graph structure of a DFA-based hypermedia document. Each box is a state, containing specifications of the objects to be displayed when the automaton is ‘in’ that state. The arrows are transitions.

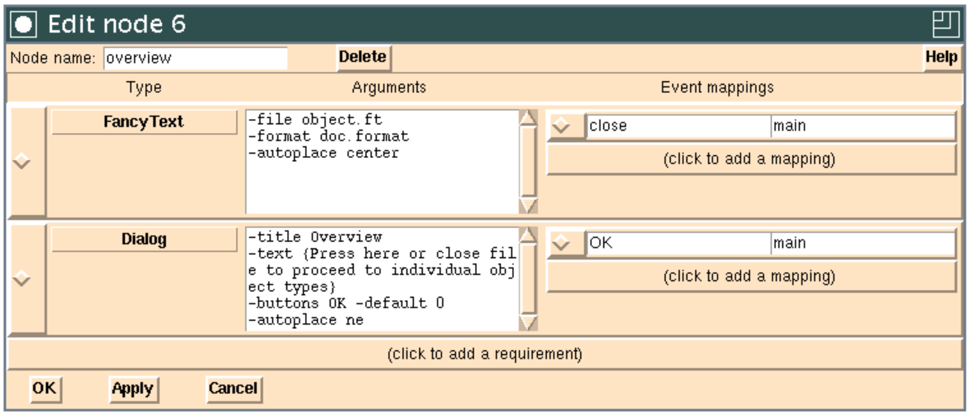

Figure 5: Editing a DFA state (the ‘overview’ node in Figure 4) with two object specifications. Each specification is represented by a panel running from left to right, showing the object type, argument list, and event mappings. The mappings translate events posted by the objects into state transitions on the graph. In this case, whether the user presses OK on the dialog box or closes the FancyText window, the state of the model will proceed to ‘main’.

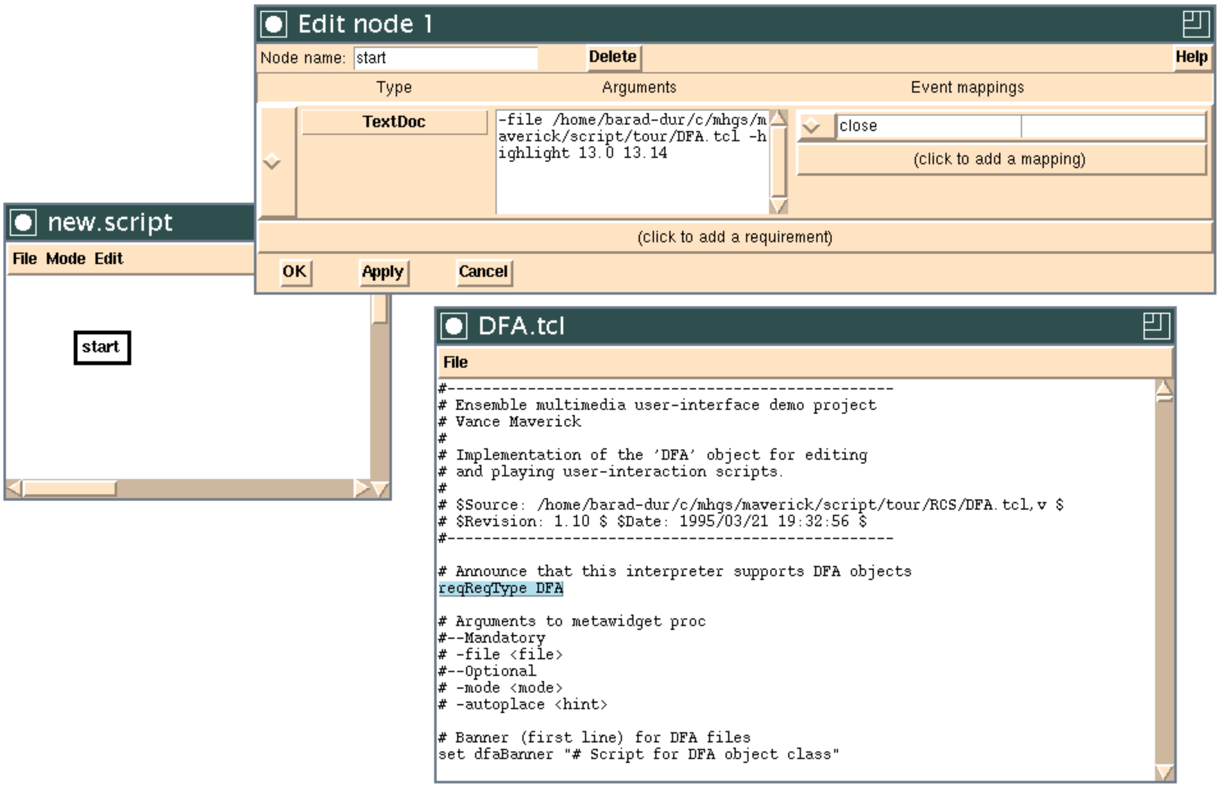

Figure 6: Programming by demonstration. The author is creating a DFA script which will reference the source file DFA.tcl. The reference is created by browsing the file and selecting the region to be highlighted, then pressing a key combination to send the appropriate object specification to the script.

The object specifications of a node in the graph can be edited through the interface shown in Figure 5. The user selects an object type from a menu button at the left of each panel, and enters the arguments for the specification. With these, the dialog calls a registry service to get the list of possible events which the object will post. (This call is simply forwarded to the corresponding object type proc.) For each possible event, the user enters an event mapping: the name of the state in the graph, if any, to which the DFA should move when that event is posted.

3.3.2 Programming by demonstration

Entering object specifications by hand is laborious and error-prone. Accordingly, we have provided a simple form of 'programming by demonstration' [19], in which the user creates object specifications by manipulating a sample object directly. When it is in the desired state, a standard user-interface action (in our examples, a click with mouse button 3) causes the object to broadcast the specification describing it. This broadcast event may be caught by the script and incorporated. Figure 6 shows this authoring mechanism in action.

3.4 Interoperability

While the hypertext and DFA objects implement different hypermedia models, they are perfectly compatible. The DFA in Figure 4, for instance, brings up several different hypertext objects during its operation. The web of information reached from such a hypertext object becomes an extension of the DFA state that required it; when the DFA leaves the state, the web is cleared away. Similarly, one of these hypertext objects brings up another DFA (as a demonstration of DFA programming); this object and any others it requires are dependent on the hypertext object.

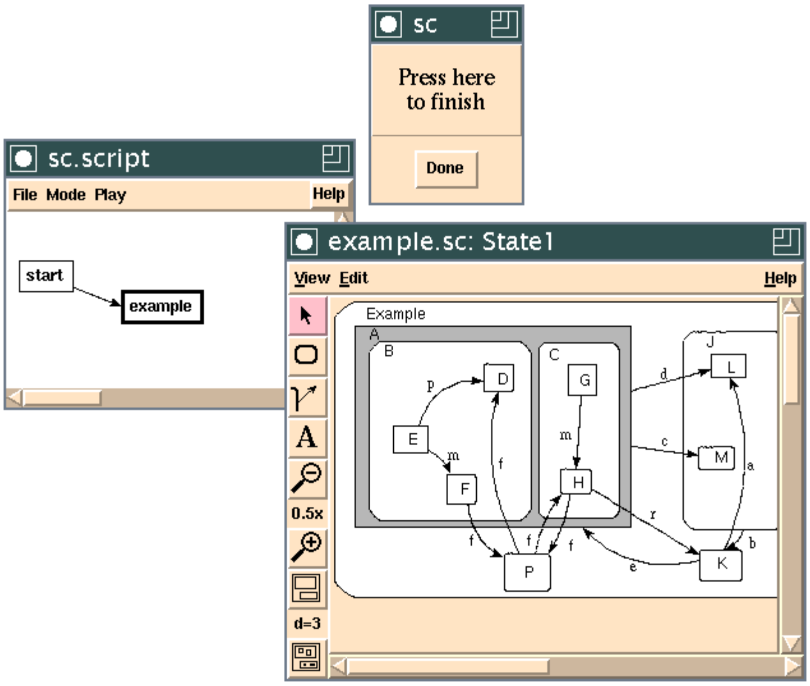

Figure 7: Integrating external applications with Jicama. Here a DFA script brings up a StateCharts program via `sc`, an application running in a separate process.

3.5 Related work

This DFA model is clearly related to the browsers of NoteCards [14], and to the Petri nets of Trellis [10]. Each of these models could easily be implemented in Jicama. A NoteCards browser consists of pointers to other objects, and of links which are to be added to the objects when they come up. This model would require a new object argument convention for the insertion of links at display time. A Petri net model could be implemented with Jicama and its object conventions as they stand; the main difference between it and our DFA object would be in its handling of events.

4 Porting Applications to Jicama

Jicama is an integration framework: its purpose is to make it easy to integrate diverse applications and data types into scripted hypermedia documents. Up to now, we have demonstrated it only with specially developed object types. An important test of the system is the ease with which already existing applications may be adapted to it.

To carry out this test, we chose `sc` [6], an editor for the graphical language StateCharts [15]. It was of interest to us for several reasons. First, it is fairly large: 3593 lines of Tcl code. Second, it is written in an object-oriented dialect of Tcl, `[incr tcl]` [17], so it tests the effectiveness of Jicama in integrating applications built on separate, specialized bases. Finally, it allows users to edit one program in several views at once — an important feature when graphical programs can grow beyond the limits of a single window. For us, this feature suggested that if a 'guided tour' included links to several regions of the same StateCharts program, the Jicama requirement protocol would permit `sc` to display them both, in one window or two depending on the distance between them.

Figure 7 shows `sc` running under control of a DFA script, using Jicama. To integrate it, we added about 50 lines of Tcl code, some of it "boilerplate" copied from a template implementation of the object interface; and a standard file of 58 lines of C code, in which only 5 lines needed to be edited. There was no difficulty in using Jicama with `[incr tcl]`. And since `sc` remained a separate application, we were able to run it on a fast remote processor connected to the same X server as our scripting tool.

5 Future Work

There are two main directions in which Jicama will develop. First, it will be integrated into Ensemble [5], a multimedia document system and software development environment. Second, it will grow to include a mechanism for synchronization of continuous-time media, such as video and sound.

Incorporation of Jicama into Ensemble will help us address one of the pressing problems of most hypermedia systems, including the demonstration systems created for Jicama: the difficulty of overseeing the structure and content of a highly connected document. Its multiple presentations and multiple views [13] will enable the construction of 'linearized' views of graph-structured documents. And its support of derived or computed documents will allow the maintenance of consistent 'overviews' of scattered material.

Several of the issues raised by the designers of NoteCards in their retrospective paper [14] are already addressed in Ensemble. Its current design, for instance, provides a clear distinction between *inclusion* and *reference* links, as they advised. Further, its sophisticated versioning system [24] allows secure, fine-grained recovery and review of information from any point in the history of a document.

With most objects running in the same Ensemble process, it will be easier to provide the synchronization necessary for sophisticated scheduling of continuous media, such as that provided by Firefly [4]. It remains an interesting research problem, though, to define a lightweight integration framework for separate applications, along the lines of Jicama, which will permit such synchronization and scheduling.

6 Conclusion

In this work, we set out to develop a framework for integrating diverse object types developed under Tcl/Tk into scripted hypermedia interactions. We have presented a simple design, Jicama, based on a run-time representation of the relationships between objects as a directed graph. In this framework, it is easy to write new compatible object types and to integrate existing applications. They may be controlled by one of the two hypermedia models we have implemented in Jicama; but it is also easy to implement new hypermedia models, including many of those described in the literature. Diverse models are fully interoperable.

7 Acknowledgments

We drew on code written by several other people: Boris Vaysman's Tk video widget, the core of the Jicama Video object; John Sefler's `draw`, which creates the files read by the Graphics object; and `sc`, written by Stephen Edwards.

Brian Dennis, Ethan Munson, Tom Phelps, and Tim Wagner gave useful advice on the writing of the paper.

References

[1] Kenneth M. Anderson, Richard N. Taylor, and E. James Whitehead, Jr. Chimera: Hypertext for heterogeneous software environments. In *European Conference on Hypermedia Technology*, pages 94–107, Edinburgh, Scotland, 1994. ACM.

[2] Tim Berners-Lee, Robert Cailliau, Jean-François Groff, and Bernd Pollermann. World-Wide Web: The information universe. *Electronic Networking: Research, Applications, and Policy*, 1(2):52–58, Spring 1992. Available via `ftp://ftp.w3.org/pub/www/doc/ENRAP_9202.ps.Z`.

[3] Tim Berners-Lee and Daniel Connolly. *Hypertext Markup Language: Internet Draft.* CERN, Geneva, Switzerland, 1.0 edition, July 1993. Available via `ftp://ftp.w3.org/pub/www/doc/html-spec.ps.Z`.

[4] M. Cecelia Buchanan and Polle T. Zellweger. Specifying temporal behavior in hypermedia documents. In *European Conference on Hypermedia Technology*, pages 262–271, Milan, Italy, 1992. ACM.

[5] Brian M. Dennis, Roy Goldman, Susan L. Graham, Michael A. Harrison, William Maddox, Vance Maverick, Ethan V. Munson, and Tim A. Wagner. A document architecture for integrated software development. Available via `ftp://ftp.cs.berkeley.edu/ucb/mhgs/maverick/ensemble.ps`, 1994.

[6] Stephen Edwards. *An Interactive Editor for the Statecharts Graphical language*. University of California, Berkeley, 1994. `http://www-cad.eecs.berkeley.edu/~sedwards/statecharts/index.html`.

[7] Douglas C. Engelbart. Toward integrated, evolutionary office information systems. In *26th Joint Engineering Management Conference*, pages 63–68, Denver, Colorado, 1978. IEEE.

[8] Douglas C. Engelbart and W. English. A research center for augmenting human intellect. In *Fall Joint Computer Conference*, pages 395–410, San Francisco, California, 1968. AFIPS Press.

[9] Andrew M. Fountain, Wendy Hall, Ian Heath, and Hugh C. Davis. Microcosm: An open model for hypermedia with dynamic linking. In *European Conference on Hypermedia Technology*, pages 298–311, Versailles, France, 1990. ACM.

[10] Richard Furuta and P. David Stotts. Programmable browsing semantics in Trellis. In *Hypertext '89 Proceedings*, pages 27–42, Pittsburgh, Pennsylvania, 1989. ACM.

[11] Brian R. Gaines and Mildred L. G. Shaw. Open architecture multimedia documents. In *ACM Multimedia 93: Proceedings of the First ACM International Conference on Multimedia*, pages 137–146, New York, NY, August 1993. ACM Press.

[12] Charles F. Goldfarb. *The SGML Handbook*. Oxford University Press, Oxford and New York, 1990.

[13] Susan L. Graham, Michael A. Harrison, and Ethan V. Munson. The Proteus presentation system. In *Proceedings of the ACM SIGSOFT Fifth Symposium on Software Development Environments*, pages 130–138, Tyson's Corner, VA, December 1992. ACM Press.

[14] Frank G. Halasz. Reflections on NoteCards: Seven issues for the next generation of hypermedia systems. *Communications of the ACM*, 31(7):836–852, July 1988.

[15] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[16] Joel McCormack, Paul Asente, and Ralph R. Swick. *X Toolkit Intrinsics – C Language Interface*. MIT X Consortium, Cambridge, MA, 1994. Version 6 (Donna Converse, ed.), distributed with the X Window System.

[17] Michael McLennan. `[incr tcl]`. AT&T Bell Laboratories, Allentown, Pennsylvania, 1993. Version 1.5, available via `ftp://ftp.cs.columbia.edu/archives/tcl/contrib/extensions/itcl-1.5.tar.gz`.

[18] Microsoft Press, Redmond, WA. *OLE 2 Programmer's Reference*, 1994.

[19] Brad A. Myers. *Creating User Interfaces by Demonstration*, volume 22 of *Perspectives in Computing*. Academic Press, Boston, Massachusetts, 1988.

[20] John K. Ousterhout. An X11 toolkit based on the Tcl language. In *Winter Usenix Conference*, 1991.

[21] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, Massachusetts, 1994.

[22] Steven P. Reiss. Connecting tools using message passing in the Field environment. *IEEE Software*, pages 57–66, July 1990.

[23] Randall H. Trigg. Guided tours and tabletops: Tools for communicating in a hypertext environment. In *Conference on Computer-Supported Cooperative Work*, pages 216–226, Portland, Oregon, 1988. ACM.

[24] Tim A. Wagner and Susan L. Graham. Integrating incremental analysis with version management. Submitted to the European Software Engineering Conference, 1995.

[25] Nicole Yankelovich, Bernard J. Haan, Norman K. Meyrowitz, and Steven M. Drucker. Intermedia: The concept and the construction of a seamless information environment. *IEEE Computer*, 21(1):81–96, January 1988.

[26] Polle T. Zellweger. Scripted documents: A hypermedia path mechanism. In *Hypertext '89 Proceedings*, pages 1–14, Pittsburgh, Pennsylvania, 1989. ACM.

A Tcl Interfaces of Jicama

In these interfaces, objects and requirements are always passed by handles. An object handle is a unique string. A requirement handle is a list, containing two object handles and a unique integer ID.

A.1 Object registry server

`reqAddMaster` *obj type*	Add the given object to the list of permanent objects.
`reqDeleteObj` *obj* {*evt* {*arg ...*}}	Mark the object as deleted, posting the given event (default `close`).
`reqAssert` *obj spec*	Assert the given requirement (return is a requirement handle).
`reqRelease` *owner obj*	Release all requirements between the two objects.
`reqRelease` *req*	Release the specific requirement identified by the handle.
`reqSweep`	Trigger a garbage collection, releasing any unneeded objects.
`reqRegType` *type* {*type ...*}	Inform the registry that the current application supports the given types.
`reqTypeList`	Get a list of all the existing types.
`reqTypeEvents` *type args*	Get the list of events which can be posted by an object of the given type created with the given arguments.
`reqPostEvent` *obj evt* {*args*}	Post an event to all objects which require this one.
`reqBroadcast` *evt* {*args*}	Post an event to all broadcast clients.
`reqBroadcastClient` *obj*	Make the given object a broadcast client.
`reqInfo`	Print a debugging dump of the object-requirement net.

A.2 Object type proc

Each object type proc (called *type* below) must be registered with `reqRegType`.

type `satisfy` *alist* {{*obj alist*} ...}	Get an object, new or old, which can satisfy the given argument list.
type `display` *obj* {*alist ...*}	(Re)display the given object according to the given list of argument lists.
type `release` *obj*	Object is being released by the Jicama system. Carry out any necessary cleanup.
type `callback` *obj announcer args*	Event callback from another object, specified by requirement handle.
type `broadcast` *obj announcer args*	Broadcast callback from another object.
type `event-list` *args*	Return the list of events which an object of this type can post if created with the given arguments.