EClerk Office Assistant

by

Jonathan Eric Wolfe

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2004

Author.....
Department of Electrical Engineering and Computer Science
September 10, 2004

Certified by.....
Seth Teller
Associate Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by.....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

EClerk Office Assistant

by

Jonathan Eric Wolfe

Submitted to the Department of Electrical Engineering and Computer Science
on September 10, 2004, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

For decades, people have continued to collect an inordinate amount of paper documents containing important information that should be easily accessible. This paper clutter inhibits indexing this information and easily searching through it. This thesis presents the code architecture and user interface design of the Electronic Clerk, a proof-of-concept electronic office assistant. The Electronic Clerk (EClerk) is a device to assist in reducing paper clutter in the office environment. The device takes paper and speech as input, performs data binding between input streams in order to attach metadata to each document, and structures the data using the Resource Description Framework (RDF) standard. The hardware structure of EClerk consists of a dedicated computer, video camera, scanner, touchscreen, and microphone for capturing input. The software structure consists of the Galaxy speech recognition system, the Haystack information client for retrieval and modification of the collected data, optical character recognition, and a graphical user interface that provides continuous feedback to the user. Primary design principles for this device include providing continuous user feedback and robustness to imperfect input in order to provide a truly usable system.

Thesis Supervisor: Seth Teller
Title: Associate Professor of Computer Science and Engineering

Acknowledgments

I would like to thank my thesis advisor, Seth Teller, for his creative ideas and energy throughout the entire development of this thesis. His vision and support has made this project possible. I would also like to thank everyone that assisted in the integration of Galaxy and Haystack, including David Karger, Jim Glass, Scott Cyphers, and Vineet Sinha.

Special thanks also go to my hardware lab colleagues, Roshan Baliga and Kevin Wang, for all the fun times and thesis reviews.

Finally, many thanks to friends and family for the unwavering support through years of working hard and playing hard.

Contents

List of Figures

Chapter 1

The EClerk Vision

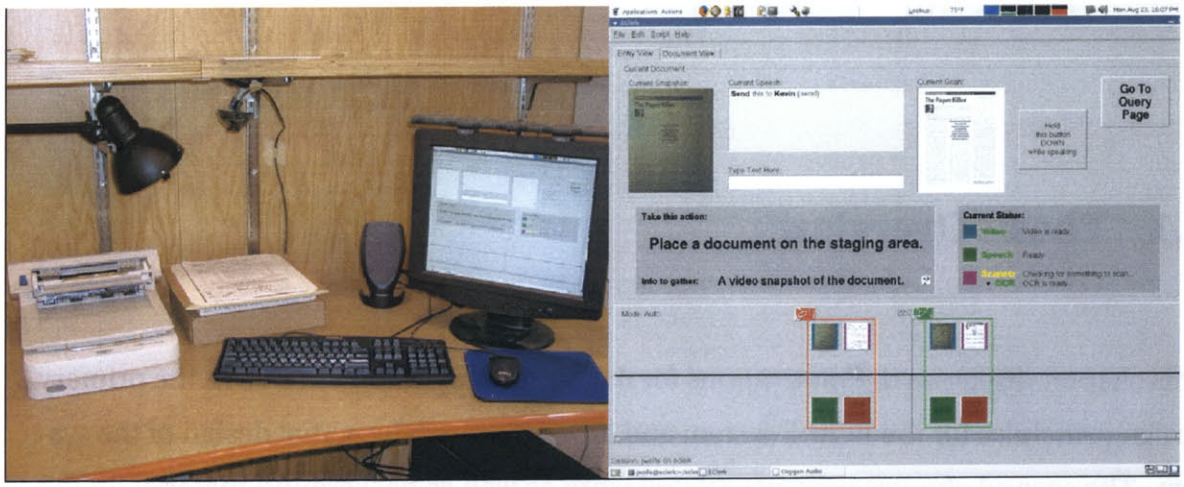

Figure 1-1: EClerk: The physical system and the user interface in action.

The Electronic Clerk (EClerk) is a prototype device to assist in reducing the clutter of documents, notes, and other paper components of the office area. Input to the device consists of these paper documents in addition to spoken or typed information from the user. This metadata is associated with other information gathered about the document to create a loose collection of data items to represent the electronic version of that paper document. Information about these documents can be retrieved later by user-issued queries through speech or the keyboard [17].

Not only will the EClerk help to organize piles of paper that would otherwise be haphazardly strewn around the office, it will provide a powerful means to query for information both specific to one document (i.e. text searches) and between many documents (i.e. all memos from a specific person). The EClerk will also deploy "agents" to process appropriate documents – such as sending an email notification to the appropriate person – by interpreting the metadata for each document and applying heuristics to determine when these agents should be invoked.

EClerk aims to couple existing technologies in speech recognition and information storage with extreme usability in user interface to make this concept a reality.

1.1 Thesis Overview

This chapter discusses the motivations for the creation of EClerk, as well as the primary design principles for the EClerk system and an overview of the actual design of the system. The second chapter presents related work as well as background material on the pre-existing components used in EClerk. The third chapter outlines the design principles used throughout the development of the EClerk system. The fourth chapter focuses on the architecture of the system itself, including a discussion of the interrelating hardware and software components that make up EClerk. The fifth chapter discusses the design of the EClerk control software and the choices made in the structure of this software component. The sixth chapter focuses exclusively on the user interface design, a key contributor to the success of the system. The seventh chapter presents results on the usability of the system. The eighth chapter outlines future work, and the ninth chapter concludes by detailing contributions.

1.2 The Need for Reduced Clutter in the Workplace

The workplace is inherently filled with paper, from memos to invoices to notes. Most of these workplace documents do not come in a bound, preorganized volume – they are typically single sheets of paper. The inevitable buildup of these documents not only clutters the workplace environment, but makes organization and cross-referencing of that disparate information much harder without an elaborate and complex paper filing system (or an administrative assistant with 4 hands).

However, since the clutter of paper is still pervasive in office environments today, it is clear that no alternative system has been successful enough to replace it. Current efforts to declutter the office environment focus on the psychological instead of the technological, encouraging office dwellers to make use of proper filing techniques and filing equipment. Even remarkably successful products like Adobe Acrobat that aim to make paperless environments ubiquitous have not eliminated the paper clutter in offices. The need still exists for a convenient, effective means to reduce this clutter in the office.

1.3 The Need for A Usable System

Everyone subject to this clutter of paper has a unique way of dealing with it to maintain order and structure. For a replacement system to be adopted, it must be as easy to use as handing a document to a human assistant with a few verbal instructions. Such a system would collect the document and the verbal metadata, process the combination on its own (as would a human), and undertake the appropriate action for that document – filing it away, sending an email, etc.

It is *imperative* that the system always be attentive and be able to batch operations in order to respond immediately to user input. In addition, EClerk focuses on *spoken* input from the user to remove the keyboard from the input process as much as possible. Depending on typed input from the user would most likely decrease productivity of document input to the point where any benefit of decluttering would be lost.

1.4 Design Principles

Following are the key principles used in the design of the EClerk system to provide such a usable filing alternative:

- The system should always provide continuous feedback to the user about whatever it is doing.
- The system should operate with enough speed to allow for an input stream of many documents with minimal lag between documents – it should batch operations well.
- The system should be robust enough to recover from imperfect user utterances or queries in order to preserve appropriate document-metadata bindings.
- The system should use a modular approach to storing the documents in a repository in order to allow for mobile storage and off-site queries.

1.5 Usage Scenarios

- Administrative Assistant received urgent mail for Professor who is traveling around the world for the next two weeks. He enters the mail into EClerk, where each piece of mail becomes a virtual document viewable as text and low- or high-resolution image. Professor securely connects to any machine which replicates the document store and issues queries on that remote database using a local copy of the Haystack interface, retrieving all new documents in the store. Professor now has access to his paper mail from anywhere, presorted by EClerk using metadata entered by Assistant.
- Unorganized Grad Student receives many paper invoices from all the companies that provide parts for the underwater vehicle system being built in lab. He enters all those invoices into EClerk as they arrive because he knows he will lose the

paper copies. The invoice is automatically filed in the document store for later retrieval, and EClerk earmarks it as an invoice because it closely resembles previous invoices and Grad Student said the work "invoice" when describing the document. Administrative Assistant gets an email from EClerk that an invoice has been entered, views the electronic copy of the invoice, and issues a reimbursement check to Grad Student.

- Professor is giving a talk in a lecture hall in MIT's new Stata Center and presents the poster advertising the talk to EClerk. EClerk marks this document as an announcement and registers the document with the Oxygen Kiosk network that displays information to people in the Stata Center lobby. Visitors to the Stata Center that are lost in the lobby query an Oxygen Kiosk for talks happening that day. The query is successful and an image of the poster is displayed to the visitors. Metadata about the location of the talk, gathered from the document by EClerk, is also provided to the kiosk. The kiosk provides directions to the visitors for how to get to the lecture hall.

Chapter 2

Background

2.1 Related Work

EClerk strives to assist human beings by interacting with and understanding the natural workflow of the user, especially speech interaction as both an input method for the user and a communications tool. The goal of this more natural interaction, present in computer system design from early on, becomes even more realizable today with new technologies for speech processing and the impetus to explore multimodal interaction with human users.

2.1.1 Memex

The Memex, proposed by Dr. Vannevar Bush in 1945, first presented the notion of a device for storing and easily retrieving personal information using associative search. Conventional systems for filing and retrieving information depended on systematic techniques such as filing alphabetically – systems that, according to Bush, did not line up with the way the human brain connects information together.

The Memex was envisioned as a device built into a desk that stores and retrieves information using associative trails. These associative trails attempt to mimic the human brain's mechanism for connecting together kernels of information. The user can

view books, communications, and anything entered into the Memex via microfilm by entering a code from a codebook that acts as an index on the repository. A projection system displays the microfilm content to the user. Connections can be made between these data by requesting a new code that links objects together into an associative trail. The same control that browses through a single item (i.e. paging through a book) can be used to navigate through such an associative trail. Facsimiles of anything stored in the system can be requested by the user in order to transport information or provide information for import into a different Memex [1].

Later proposals of the Memex make use of magnetic tape instead of microfilm as a storage medium to allow for deletion. Other additions include the use of color for distinguishing between associative trails and the use of speech technologies for speaking to the system as a form of input and control. The Memex laid the ideological groundwork for hypertext and for annotation-based information storage and retrieval [2].

2.1.2 Memory Assistant

The Software Laboratory at Hewlett Packard conducted research relating to electronic assistants. The product of this research consists of a memory assistant that allows the user to edit and query a database of storage locations for various objects around the house [3]. The off-the-shelf speech recognizer required training on the specific domain, including objects, locations, and attributes.

Due to the nature of being knowledgeable about storage locations, the domain of the recognizer for this application is inherently bound to the attributes and relationships governing the objects in question. Consequently, adding new attributes and relationships requires retraining the recognizer, making it difficult for the end-user to customize or generalize use of the system quickly and easily. However, passing the speech recognition computation to a dedicated server separate from the handheld computer to which the user makes queries provides for an elegant mobile assistant. The EClerk system attempts to uncouple input methods and computation in addition to providing a speech

domain that is painless to modify.

2.1.3 Remembrance Agent

Bradley Rhodes and Thad Starner at the MIT Media Lab constructed a system that plugs into the Emacs editor and performs a continuous associative search on user's email and notes as well as on-line documents in order to provide the user with informative suggestions based on the user's context [14]. This system attempts to use the associative nature of its search to provide answers to questions that the user does not need to pose explicitly. This sort of associative search is a successful way to make connections between the user's information and generalize based on that data.

However, suggestions made by the Remembrance Agent system currently are dependent upon the user's currently existing information sources such as email or personal notes – it does not couple the gathering of information with "smart" agent behavior. EClerk acts as both the conduit for user input and the agent to provide functionality to the user after input through these active agents.

2.2 Pre-existing Components of EClerk

2.2.1 Galaxy for Speech Recognition

Galaxy is a speech recognition package built by the Spoken Language Systems lab at MIT's CSAIL. Its modular design allows for it to be deployed either in a distributed or central manner, providing the decentralized speech solution for EClerk [15]. Galaxy removes the dependence on a particular high-powered machine for speech functionality and assists in transporting the EClerk service to multiple locations.

In addition, Galaxy provides a convenient web interface called SpeechBuilder for defining a specific domain of recognition in order to greatly increase recognition accuracy. With SpeechBuilder, system administrators can easily modify the actions and

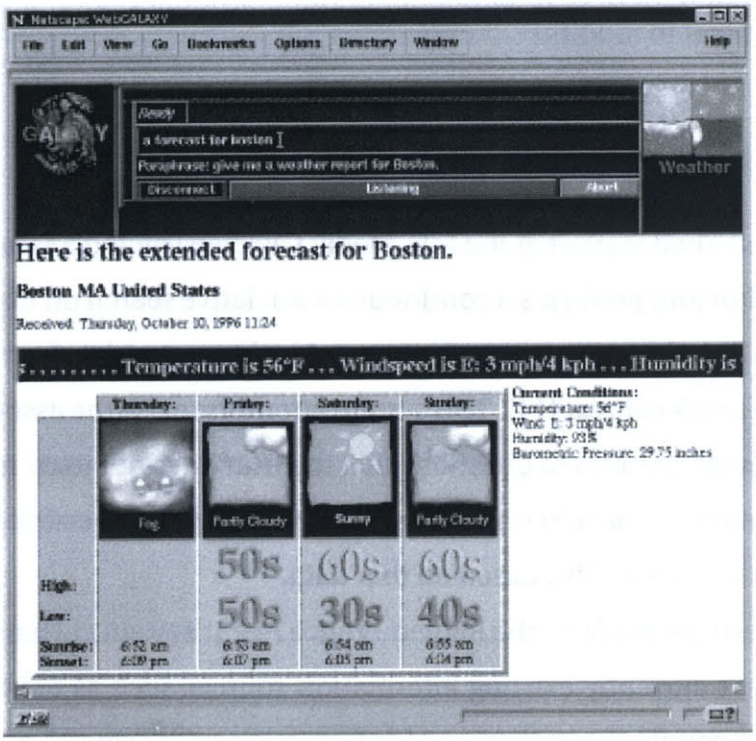

Figure 2-1: A Galaxy web interface for displaying weather information [8]. Speech recognition output and system status displays at the top, and weather information displays at the bottom.

attributes of a domain, generalize example sentences using classes of actions and attributes, compile that new domain offline, and plug in the new domain to Galaxy. Such an update need not be paired with any other system retraining – because of the limited domain of the speech recognition, Galaxy need not be trained to recognize a specific user's voice.

Galaxy is currently deployed successfully as the speech recognizer for phone systems that provide weather information and flight information, and provides the speech solution for the Oxygen Kiosks in the Stata Center.

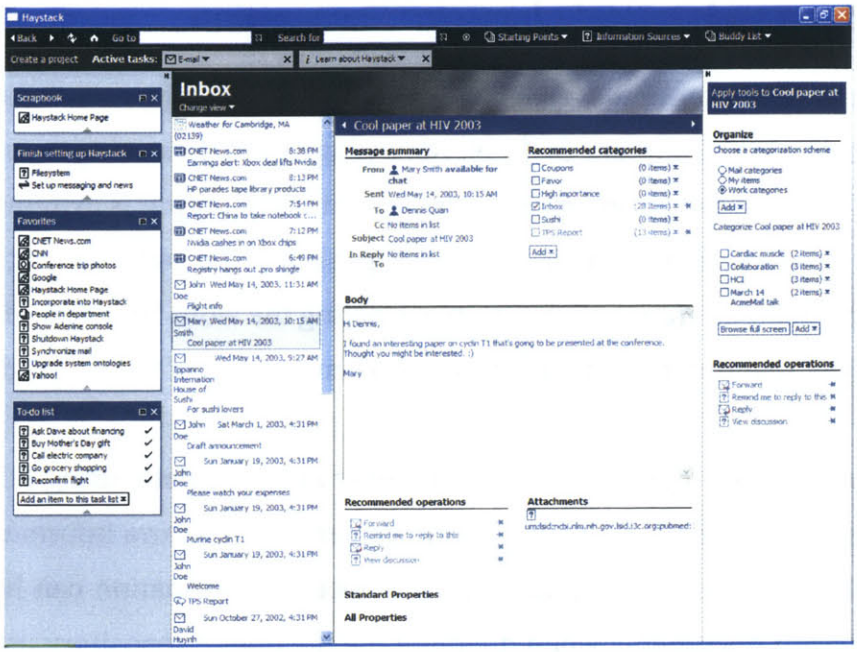

Figure 2-2: Haystack User Interface

2.2.2 Haystack for Information Storage and Retrieval

Haystack is an information retrieval client built by the Information Theory group at MIT's CSAIL. Written in Java for both UNIX-based and Windows-based operating systems, Haystack is built on a semi-structured database using the Resource Description Framework (RDF) model (`http://w3.org/RDF/`). Haystack aims to be a universal information client, using the semi-structured data model to allow users to make connections between any and all types of data ([6]).

EClerk uses this database to store data derived from user input in a data store which is already designed to effectively store and query on such semi-structured data [7]. In addition, remote instances of Haystack can make use of the same underlying RDF database to provide convenient data access and querying from many different locations.

2.2.3 Scanning and Optical Character Recognition

EClerk makes use of a scanner that is not already coupled with optical character recognition (OCR) technology. Different OCR solutions plug into EClerk using the same simple interface – input an image and receive text output. OCR output is extremely useful for annotating a document as a particular type, especially if the user does not explicitly specify that type in an utterance. For example, locating the word "invoice" in a prominent place on the scanned document gives a good indication of what that document is.

OCR is also useful for reproducing the document at a later date, using PDF format where possible. Other proprietary output formats can provide extra information such as spatial locality of words in a document. This locality information can help to retrieve more relevant documents when querying the document repository using many search terms, similar to tricks used by Internet search engines like Google. As discussed by MIT researcher and doctoral candidate Simson Garfinkel in a recent Technology Review article, OCR technology has evolved to the point where it can successfully interpret complicated documents such as bank statements with few errors, including fine print, company names, and dollar amounts [5].

New scanner technologies from Xerox and others now couple OCR capability with the scanner, including software that improves readability of the scanned image and automatically orients images and text vertically in the case of slightly rotated images [19]. Xerox, Fujitsu, and Hewlett Packard all have scanner-software packages that output searchable PDF files, allowing users to transport, search, and print documents with a single solution [9]. These technologies have the potential to simplify the task of EClerk by abstracting away the process of collecting an image in addition to an accurate textual representation of that image.

Chapter 3

Design Principles

3.1 Provide Continuous Feedback

The system should provide continuous feedback to the user about whatever it is doing. Because the system depends heavily on input from the user through many different streams including speech and video, EClerk should always provide status for which input streams it is currently processing and what it is expecting next from the user.

EClerk always displays what information it would like to gather next and what is expected of the user. In addition, EClerk displays the current status of each input stream – whether that input device is ready for input or currently processing input. EClerk also makes use of text-to-speech functionality and speaks instructions and status to the user in a responsive way. For example, whenever EClerk is given a piece of user input, it speaks the phrase "Got it." to the user. In this way, EClerk performs as much as possible like a human clerk responding in real time.

3.2 Defer Non-Critical Operations Whenever Possible

The system should operate with enough speed to allow for an input stream of many documents with minimal lag between documents – it should batch operations well by de-

ferring non-critical operations whenever possible. This goal is critical to the usability of the system: EClerk must constantly defer to user input just like a human assistant. Any latency in response frustrates human users and effectively wastes the time that EClerk is trying to save in the first place.

EClerk allows itself to be interrupted in order to absorb all user input in an asynchronous fashion. For example, EClerk will never attempt to speak instructions to the user while the user is speaking, and will cut itself off when the user provides the input that EClerk is asking for. In this way, EClerk operates like a human clerk collecting each document in turn along with the appropriate metadata to process later.

3.3 Be Robust to User Input

The system should be robust enough to recover from imperfect user utterances or queries in order to preserve appropriate bindings between document and metadata. Using various techniques like ordering constraints and optical character recognition (OCR), out-of-order scans and user utterances can be matched up heuristically with the appropriate image or speech input pertaining to that specific document. In addition, by providing continuous feedback of all explicit bindings through the user interface, the system will provide a means for the user to correct improper bindings.

Much of this goal can be accomplished through the SpeechBuilder interface and appropriate image-matching techniques. SpeechBuilder allows for example sentences that match action-type and document-type across the domain – the ability to generalize by type. By specifying a collection of these example sentences using different sentence structures, it becomes much easier to parse and accept many possible spoken inputs. In addition, rectifying low-resolution images and matching them with high-resolution images is a fairly reliable method to properly order image input and bind together images that represent the same physical document.

3.4 Provide Modularity and Mobility

The system should use a modular approach to speech recognition and storage of documents in a repository in order to allow for separate compute servers, mobile storage, and off-site queries. The design of Galaxy and Haystack lend themselves to this goal remarkably well as these systems were designed with these goals in mind. This modularity will also be useful to provide security measures if the EClerk is utilized for documents and information for which query and retrieval should be limited to specific users or groups.

In addition, a document store should be as versatile as an organized stack of paper on a desk – the person responsible for those documents should easily be able to take them anywhere and show them to anyone. EClerk strives to make its document store as versatile as possible by using a semi-structured RDF database as a document store.

Chapter 4

System Architecture and Components

4.1 EClerk Design Overview

The EClerk system consists of both a collection of hardware devices to collect the physical input from the user and a collection of software tools that provide feedback to the user and allow for query and retrieval of documents entered into the system. Research goals included:

- characterizing the problem of creating such a usable system with current technologies;
- researching appropriate hardware and software to be used for the EClerk;
- designing the user interface and code architecture of the EClerk to allow for such a usable system;
- implementing the system;
- iterating the design and implementation in order to successfully integrate the EClerk into a real office environment.

4.2 Hardware Components

The core system hardware consists of a dedicated multiprocessor computer running Linux. This allows for a load-sharing scheme where intensive tasks running locally (i.e. scanning and speech processing) will not starve the user interface from the necessary CPU cycles to respond to the user whenever necessary.

Even though the system design allows for a modular structure to decouple computation from the interface and from the document store, the prototype system consists of a central machine running those separate processes. These processes communicate over TCP sockets as if on separate machines.

As EClerk is inherently a system that captures user input, it must gather that input through a collection of connected devices. The peripherals for EClerk include:

- a scanner with a document feeder to take high-resolution images of the documents. A high-resolution image is taken to use Optical Character Recognition (OCR) to extract key textual information from the document and to maintain the ability to reproduce a physical copy of the document later;

- a video camera for taking low-resolution images of the document. Low-resolution images are taken in order to provide quick visual feedback for the user. In addition, EClerk can use this image to display to the user how it has matched documents with other forms of input like speech, giving the user confidence that the system is collecting the "correct" data together. Secondary cameras can be added later to use more sophisticated video streams for recognizing when users are presenting documents to EClerk instead of using a specific document staging area;

- a directional microphone array for speech interaction with EClerk. Such a microphone array is much less intrusive than a boom microphone headset, allowing for more natural speech interaction with EClerk while maintaining high audio quality;

- a touch-screen display for giving quick visual feedback to the user and for simple

tactile interaction with the system. The display shows the low-resolution image of the document from the camera, the spoken text associated with that document, and the high-resolution image (see Figure 6-2). Binding between all forms of input is explicitly shown to the user, in addition to the actions taken for each document.

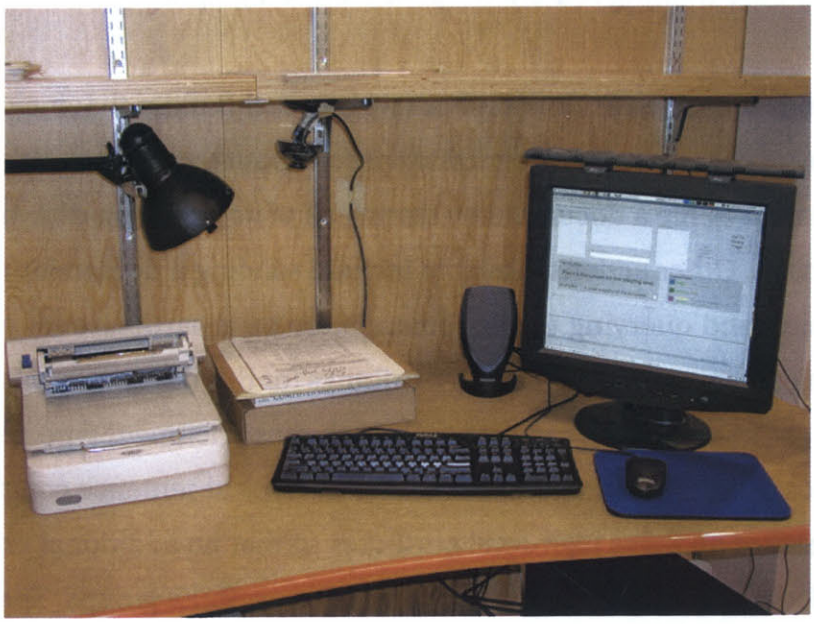

Figure 4-1: The EClerk System at MIT CSAIL

4.3 Software Components

4.3.1 Galaxy Integration

The software components of the system comprise technologies being developed within the Computer Science and Artificial Intelligence Laboratory in addition to custom software. The Galaxy speech recognition system developed by the Spoken Language Systems group (`http://www.sls.lcs.mit.edu`) forms the base of the speech recognition component of the EClerk, operating on a domain specific to EClerk's application as a clerk.

The SpeechBuilder utility, also developed by the SLS group, provides a graphical tool for modifying this domain and for invoking all of the precomputation necessary to immediately plug in the new, updated domain to the recognizer. This utility, coupled with Galaxy, allows users to abstract away the a class of objects (i.e. a "document") and therefore provide only base training sentences, greatly speeding up the process of extending the domain [18].

EClerk depends on a specific domain built with SpeechBuilder that includes the actions and attributes necessary for recognizing speech about documents. Primary action in this domain is *file*, to file a document by storing all of its pieces on disk and entering it into the database. Other actions available include sending a document to someone specific (via email) and querying the document store on a particular set of keywords (i.e. "Show me all my invoices from Dell."). Attributes for this domain consist of general types of objects or entities referenced in speech, including document type, people, companies, and times. Example sentences that define the typical sentence construction for utterances generalize on these high-level attributes so that no additional example sentences are needed in order to add any objects to these attribute groups. SpeechBuilder compiles this defined domain into downloadable files that plug directly into the Galaxy recognition system.

4.3.2 Haystack Integration

The Haystack information retrieval system (`http://haystack.lcs.mit.edu`) developed by the Theory of Computation group acts as the information retrieval client. Haystack is designed to facilitate retrieval and modification of data with a custom description in the form of RDF schemata [7]. Utilizing an underlying database built on RDF, Haystack provides both flexibility with data types and a single interface for associated tasks and applications; for example, manipulating metadata and sending email.

The underlying database for Haystack, written in C++, interacts with the world through a language called Adenine. Adenine essentially provides console access to the database

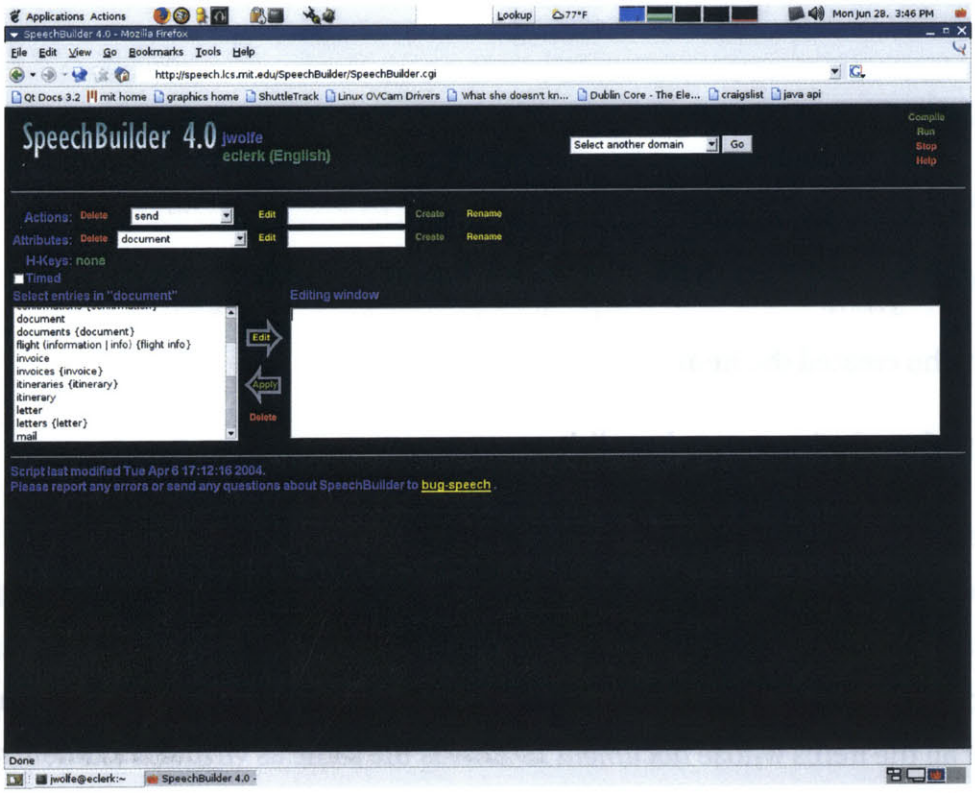

Figure 4-2: SpeechBuilder User Interface. Users can easily add actions, attributes, and example sentences. The domain can then be downloaded and plugged right into Galaxy.

with through key operations such as *add*, *remove*, *contains*, and *query*. EClerk connects to this database through this Adenine console and issues Adenine statements in order to manipulate the underlying RDF [12].

RDF triples can be formed out of *resources* (concrete objects in the database, or abstract concepts relating objects) and *literals* (typically strings) in order to provide description. A triple consists of a subject, predicate, and object. For example, the triple *(Fred, hasSister, Mary)* indicates that Fred has a sister named Mary. These triples form the basis of the semi-structured schema EClerk uses to describe documents.

In order to have a consistent representation of documents in this database, EClerk uses a specific schema to describe virtual documents. First, each piece of data that is combined together to form a virtual document consists of a collection of fields describ-

ing this segment of data. These fields are:

- Item type (high-res image, speech, etc.)
- Item name
- Creation timestamp
- User who created the item
- Full path to the item stored on disk

In order to create a virtual document, an additional field is used for each segment of data called the *document ID*. This field, which is always unique, is added to each piece of data that belongs to a particular virtual document. Virtual documents are assigned these unique identifiers upon creation. Therefore, any virtual document by definition consists of all the items whose *document ID* field is the same as virtual document's ID.

Each field above is represented in the RDF database using the Dublin Core standard. The Dublin Core schema for RDF consists of many elemental properties of documents and is already supported by the RDF standard [4]. The Dublin Core element properties *type*, *title*, *date*, *creator*, and *source* represent the fields above. In addition the *identifier* property represents the *document ID* field.

4.3.3 EClerk Control Software

The EClerk control software glues these pieces together. Code design for this control software is highly modular, based on Galaxy's modular design. Modules consist of device-specific software that handles each input stream in addition to a user interface module and a module for abstracting away stable storage. A central hub facilitates communication between modules and performs the binding between the segmented data stream input chunks (see Figure 4-3 [15]). The control software is discussed in detail in the following chapter.

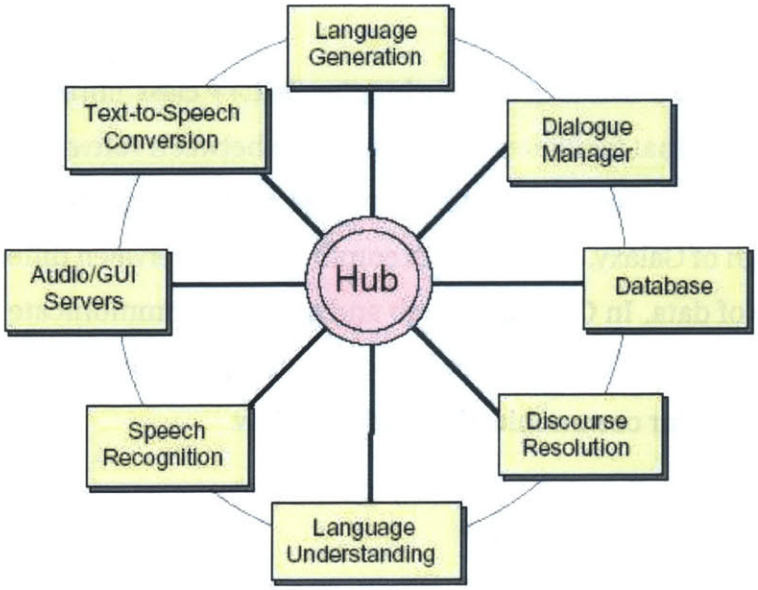

Figure 4-3: Galaxy Architecture. Modules communicate with the hub using TCP sockets.

4.3.4 Symmetric Multiprocessing in Linux

Because the structure of EClerk is inherently a collection of independent software working together to perform the function of a clerk, Linux running on a multiprocessor machine should provide a noticeable performance increase over machine with a single processor – multiple processes can be scheduled by the multiple processors to complete work faster. Since many of the tasks involved, including processing video frames from the camera and running the speech recognizer on user speech input, are CPU-bound instead of I/O-bound, such a scheme should in fact produce higher performance. It is assumed that the Linux kernel handles scheduling among the multiple processors in a manner that actually provides increased performance. Empirically, there is a noticeable increase in system responsiveness with two active processors than with one.

4.3.5 Qt

The control software as a whole is built using the Qt C++ class libraries. Qt provides an event-driven library that facilitates communication between software substructures. This library provides a synchronous callback system termed "signals" and "slots" [11]. Similar to the design of Galaxy, a central hub communicates between different modules to control the flow of data. In Galaxy, the hub and modules communicate via TCP connections (see Figure 4-3). EClerk's hub and modules use Qt's optimized callback system in order to achieve similar communication and data flow.

Chapter 5

Software Design – EClerk Control Software

5.1 EClerk Hub

5.1.1 Facilitating Module Communication

The hub's role as a communication proxy consists of accepting signals by modules into slots defined in the hub and emitting the proper signals connected to slots in particular modules. In addition to this proxy behavior, the hub also sanity checks the data being passed around in order to intercept any rogue communications initiated by malfunctioning modules.

This architecture with signals and slots provides a powerful abstraction to quickly built a user interface in an event-driven way – a task necessary for such an interface as this system depends on user input to make progress and perform useful tasks for the user. In addition, using the Qt libraries allows for software ports to any operating system that is supported by Qt with much less effort than porting code based on operating-system-dependent graphical libraries. Qt currently supports Windows, Unix/Linux, Mac OS X, and embedded Linux.

5.1.2 Performing Bindings

The hub ties the disparate input streams into collections termed "bindings". These bindings group together all the individual pieces of data collected from the input streams pertaining to a particular document entered into the system. The hub collects the individual items from the specific modules in charge of collecting the data from the raw devices and sets up these bindings.

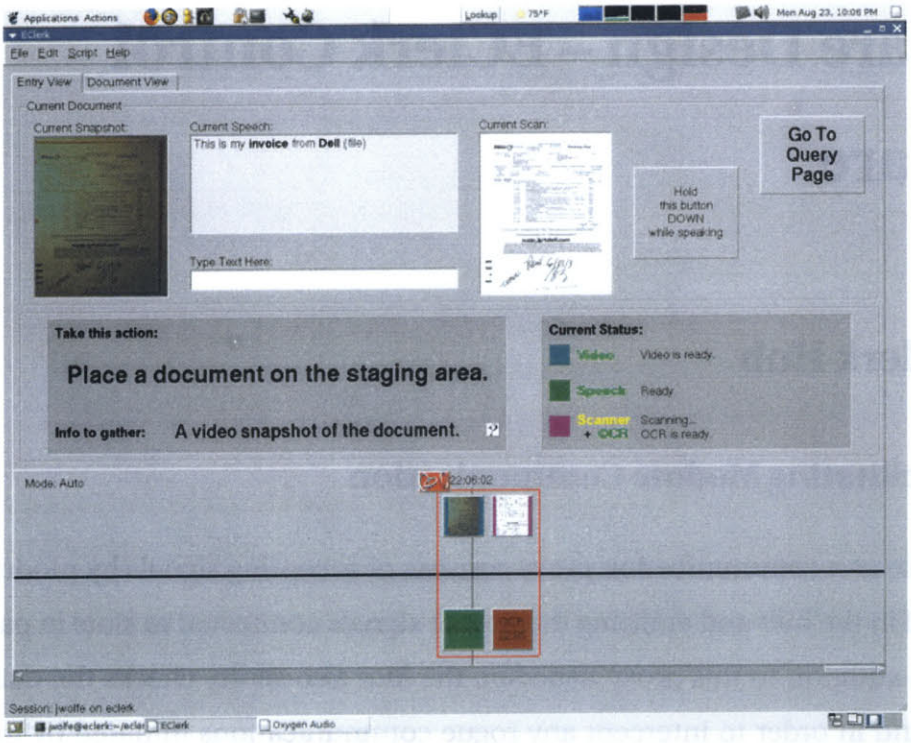

Figure 5-1: A "binding" created by grouping user input together to form a virtual document.

In the simplest of cases, a "binding" consists of one transcribed utterance, one low-resolution image, one high-resolution image, and the resulting OCR text output when run on the high-resolution image. In more complicated scenarios where more than one piece of data from a given input stream "belongs" in the binding for a particular document, the hub attempts to match up these extra items by correlating text or images with

items already existing in the binding (see Figure 5-1).

In order to correlate text, the hub takes the keywords from speech input gathered from Galaxy's parse and performs keyword matching using those words against current text items in the current binding. To correlate an image, the hub resizes the larger image and performs a simple pixel match within a given threshold for each image in the current binding. If no matching is possible, the hub assumes that the new item should belong to a new binding. The user has ultimate control of which items belong to a particular binding by dragging and dropping items between bindings.

For multipage documents, a "binding" consists of a regular binding with a collection of scan/OCR text pairs for each additional page in the document. For simplicity, the low-resolution image in a binding for a multipage document is just the first page of the document.

5.2 EClerk Modules

Modules whose purpose is to gather user input handle the device-specific nature of each mode of input – a Firewire video camera for gathering low-resolution snapshots, a flatbed scanner with a document feeder for gathering high-resolution scans, and Galaxy for gathering speech input. In addition, a module for processing the high-resolution images into text via OCR provides metadata which acts like additional user input. Output from this module is treated the same as user input, only it is invoked after each completed scan instead of triggered directly by explicit user actions.

5.2.1 Speech Module

The speech module interacts with Galaxy via 2 TCP sockets to further modularize the control of the recognition software. One socket, which communicates with the program *galaudio*, is used for control messages. The other socket, which communicates with the Frame Relay module of Galaxy, is used for semantic messages. Control messages consist

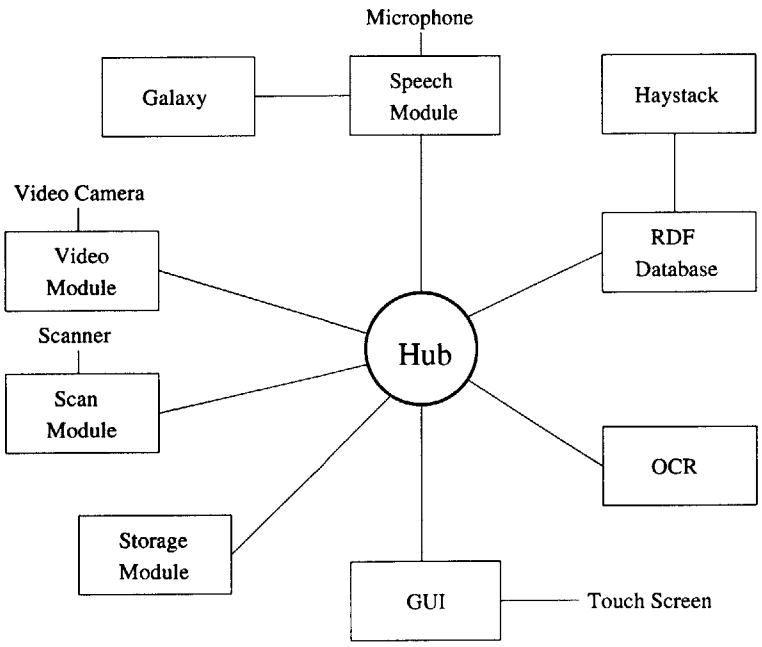

Figure 5-2: EClerk Architecture. Modules communicate with the hub using synchronous callbacks provided by the Qt class library.

of microphone on/off events and status updates, while semantic messages consist of Galaxy's parse of speech input.

If open-mic recognition is used, Galaxy determines on its own when segments of audio input are in fact considered speech input. Otherwise, microphone on/off events are used to tell Galaxy when the user is speaking. This is accomplished with a button that the user holds down when speaking to the system.

From Galaxy's parse of the spoken text, the module extracts the parse frame and provides these attribute/value pairs in addition to the recognized text to the rest of the system. For example, the *action* field may be set to "File" and the *document* field may be set to "invoice". This allows the hub to decide if any smart agent-based actions should be invoked based on the user's speech (i.e. for a "Send" action, the system attempts to make a good guess for who to send a copy of the document to via email).

In addition, these pairs allow for speech to carry semantic meaning other than just

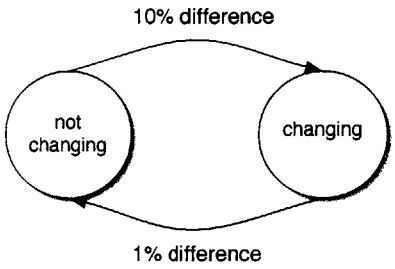

Figure 5-3: Example frame changing FSM. This finite state machine diagram shows that a larger difference in pixels from one frame to the next causes a status change to "changing" where a smaller pixel difference causes the status change back to "not changing".

document metadata. EClerk's current speech domain, built using the SpeechBuilder tool, contains a Query action and a Send action. Query actions perform a query on the document store using the recognized words from the user's speech as the keywords for the search. Send actions operate as above by sending a copy of the most recent document entered into the system to the person specified by the user (assuming EClerk knows that person's email address).

5.2.2 Video Module

The video module handles the driver-specific communication between EClerk and the video camera used to capture low-resolution images of documents. Consequently, the video module performs device initialization, captures and processes each frame from the camera, and performs some simple image manipulations to provide a packaged product (i.e. a processed image) to the rest of the system.

Each frame that the module accepts is compared to the last frame using a simple pixel-matching routine that tests if pixel values are within a certain threshold. If the frame is above the threshold for the number of pixels that are different from the previous frame, the frame is considered "changing". This status is saved for use in determining when to actually consider a frame as a "snapshot" of a document. The "changing" status is set back to "not changing" if the percentage of pixels that are different is below

a *smaller* threshold than the previous threshold used for deciding if a frame is changing. This simple hysteresis with threshold values causes a switch from "not changing" to "changing" when *more* of the frame is different than required for a change from "changing" to "not changing". Effectively, this process prevents taking duplicate snapshots if small changes occur in the field of view of the camera, while larger changes (i.e. placing a new document in front of the camera) trigger a change in status (see Figure 5-3).

After a status change from "changing" to "not changing" occurs, the module takes an initial snapshot of the frame when the status changes. This snapshot is compared against the background image saved from the very first frame the camera captures on program startup to determine if the candidate snapshot is something other than the background (it is assumed that the system is launched under consistent lighting conditions and camera orientation). Again, if the candidate image is within a certain threshold of difference from the background image, the snapshot is thrown away as a background image.

In addition, the candidate snapshot is compared to the previous successful snapshot to exclude duplicate snapshots of the same document if significant frame changing events occur without moving the document in the camera's view (i.e. waving one's hand in front of the camera).

Finally, once a snapshot is determined to be worthy of forwarding to the rest of the system, it is processed by one final routine that corrects the image for radial distortion. As we do not assume anything about the quality of the camera lens, images tend to be rather significantly radially distorted. The following assumption is made regarding the form of the radial distortion:

$$r_{correct} \approx r_{distorted} + k \cdot r^3_{distorted}$$

It is assumed that this one parameter k determines the majority of the distortion. For each pixel in the image, the module precomputes the correction value for the stored

value of k for the particular camera orientation used and stores the values in a matrix. Before export to the rest of the system, the candidate snapshot is corrected for radial distortion by computing the new radius of each pixel and assigning the appropriate pixel value to that position using the precomputed undistortion matrix. The diagnostic mode described later includes a graphical interface for adjusting the parameter k for new cameras or camera orientations.

5.2.3 Scan Module

The scan module handles communication between EClerk and the scanner attached to the system. Using the SANE drivers (`http://www.sane-project.org/`), this module polls the scanner often in an effort to be as responsive as possible to documents placed on the document feeder. The module does not perform any image processing on the high resolution images captured from the scanner.

The frequency with which this module polls the scanner is increased for a small period time (on the order of half a minute) after it receives a signal that a snapshot has just been successfully taken. The reason for this frequency increase is that documents to be scanned are often placed on the scanner's document feeder immediately after they are placed in front of the camera. This allows the module to be even more responsive to the user's typical workflow.

The module continues to scan all documents on the document feeder until the feeder has no more documents on it. Matching these images with the appropriate low resolution images is left to the hub when binding is performed.

After a successful scan, the module sends a signal that the scan has completed so that the hub or other modules can take appropriate action. Most notably, the OCR module (discussed below) takes this cue to gather as much textual information from the high resolution image as possible.

5.2.4 OCR Module

The OCR module handles the optical character recognition for high resolution images gathered by other modules.

For OCR software that supports providing location information for each word recognized, the OCR module includes a parser that caches this location information and provides responses to queries for words close to a target keyword. The parser uses simple Euclidean distance to calculate words that are close.

5.2.5 Storage Module

Other helper modules assist the hub and input modules by abstracting away common tasks for EClerk. The most utilized of these helped modules is the module abstracting stable storage. The storage module archives user input items to disk and facilitates common communication with the RDF database that connects EClerk to Haystack.

The storage module uses a consistent set of methods that are invoked by the hub in order to store collected data on disk in a consistent manner. Input for each day resides in a separate directory separated farther into different types of input. Each file representing a different piece of segmented data is timestamped using the filename.

The storage module also maintains a connection via a TCP socket with the RDF database in order to keep a consistent stable database store for the documents gathered by the system. The module issues RDF queries in syntax following the Adenine language developed for use with Haystack, and the database returns results.

Results from the database are *not* cached by the module. This design choice results from the assumption that the system should not always assume that it is the only entity operating on the database. In order to prevent stale cache entries if the user were to update some metadata using the Haystack interface itself while leaving EClerk active, the module reissues all queries in an effort to remain stateless and circumvent consistency problems.

5.2.6 Text-to-Speech Module

To provide spoken instructions and responses to the user, EClerk uses a module for generating text-to-speech output using Festival (`http://www.cstr.ed.ac.uk/projects/festival/`). This module is used by EClerk to prompt the user for the next piece of information that is expected. In addition, EClerk confirms each piece of user input with a quick spoken affirmation ("Got it."), generated with the Festival module.

This text-to-speech module complies with the original goal of batching operations well because it is interruptible by user action. Any input coming from the user during a text-to-speech event causes that event to be interrupted – EClerk will not talk over a user action, especially when that action is in response to the current instruction being given on the screen and through text-to-speech. In addition to this behavior, this module effectively shuts down for users that interrupt it repeatedly. This way, the system responds to users that already "know the ropes" of the EClerk system, allowing for more productive and less annoying sessions.

The module introduces a small lag before the on-screen instructions are mirrored using generated speech. This lag gives the user time to respond to the on-screen instructions before generating this extra response.

5.2.7 Scripting Module

For testing purposes, another helper module provides a scripting functionality for EClerk. This module records all user input during a specific period of time to be played back to EClerk at a later time. Scripting creates a portable bundle, including a backup of all the raw input data, in order to allow the saved scripts to be played back on any instance of EClerk without dependence on the disk of the computer where the script is created.

This scripting module is extremely useful for debugging the EClerk control software behavior in response to a particular sequence of user inputs and for giving demonstrations of the system (see Appendix B).

Chapter 6

Software Design – User Interface Design

6.1 Entry Mode

Entry mode is the primary mode of EClerk operation – it provides the interface for users to input information about documents to the system. Most of the effort in creating a successful interface for Entry mode lies in providing continuous feedback for all input and displaying what EClerk is doing with that input in a clear and consistent manner.

6.1.1 Displaying Current Input

One of the primary goals of the system is to always provide feedback to the user. The EClerk interface attempts first and foremost to do just that. The most straightforward way to show the user that the system is listening to the user is to echo back the information it gathers immediately after gathering it. Consequently, a large percentage of screen real estate is dedicated to this task. The top of the Entry mode interface displays thumbnail images and text for the most recent input into the system from the camera, scanner, and speech recognition streams.

For each item entered as input on any of the streams, the interface also displays a thumbnail of that item on a scrolling timeline below the current items view. These items are bound together with other items that correspond to the same physical document.

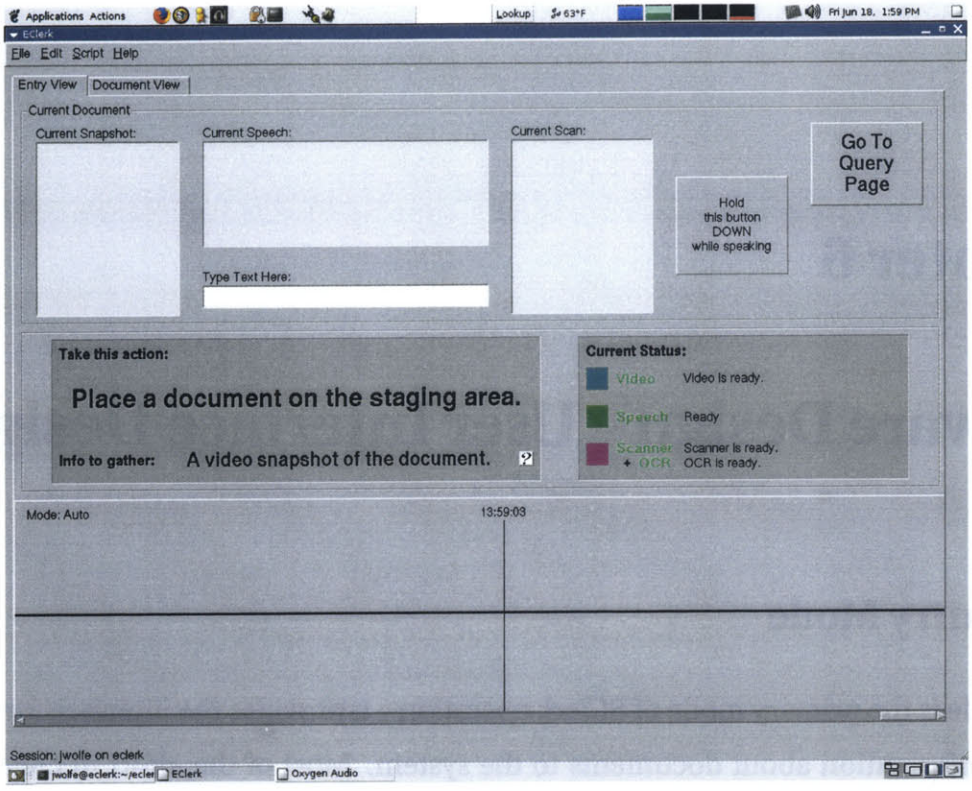

Figure 6-1: EClerk User Interface: Entry Mode. Current document overview at the top, input stream status in the middle, and a timeline of entered documents at the bottom. Input stream elements are color-coded in the timeline to indicate input element type – video snapshot, scan, etc. The input stream display indicates status using colors – green for ready, yellow for processing, and red for errors.

This binding, or more aptly this "virtual" document can be viewed by the user at any time by clicking on the thumbnail versions of any of the input items making up this new collection.

6.1.2 Displaying Current Device Status

Information that is currently being captured is as important as what information has just been captured – the user should not be left in the dark when attempting to feed input to the system. In order to show this in a consistent manner, EClerk input modules

implement an identical status interface which consists of a status string and one of two main states – "ready" and "processing". Each input module provides an appropriate string for display to the user each time its state changes. A third state, "disabled", exists for when devices are offline, unplugged, or otherwise malfunctioning. Status updates as close to realtime as possible.

EClerk displays this status information in a clear and consistent manner. Not only is the representation of stream "status" the same across all input streams using consistent colors and status strings, the user is also provided with stream-specific feedback about the current processing task – whether the system is rectifying an image or performing OCR or warming up the scanner.

6.1.3 Prompting the User

In order to tell the user what information EClerk is interested in gathering and what EClerk expects the user to do next, the system must provide some sort of prompting to the user for the next piece of input. EClerk achieves this with a combination of written instructions in a fixed place on the screen and synthesized text-to-speech instructions in addition to the written instructions.

In addition to speaking the instructions to the user, EClerk allows the user to interrupt the spoken instructions by providing input to the system by either presenting a document or speaking to EClerk. In this way, users who are experienced or do not wish to hear the spoken instructions are not continuously bothered by them. After a few repeated interruptions, EClerk will no longer provide spoken instructions, assuming that the user continues to interrupt the system because she knows the process for entering documents well already.

6.1.4 Displaying Bindings as "Virtual" Documents

Once EClerk has gathered information about a document from the user, it places that information into "bindings" as described earlier. Small views of these bindings are dis-

played at the bottom of the screen on a scrolling timeline, grouped together with bounding boxes around each collection of bound items (see Figure 6-2).

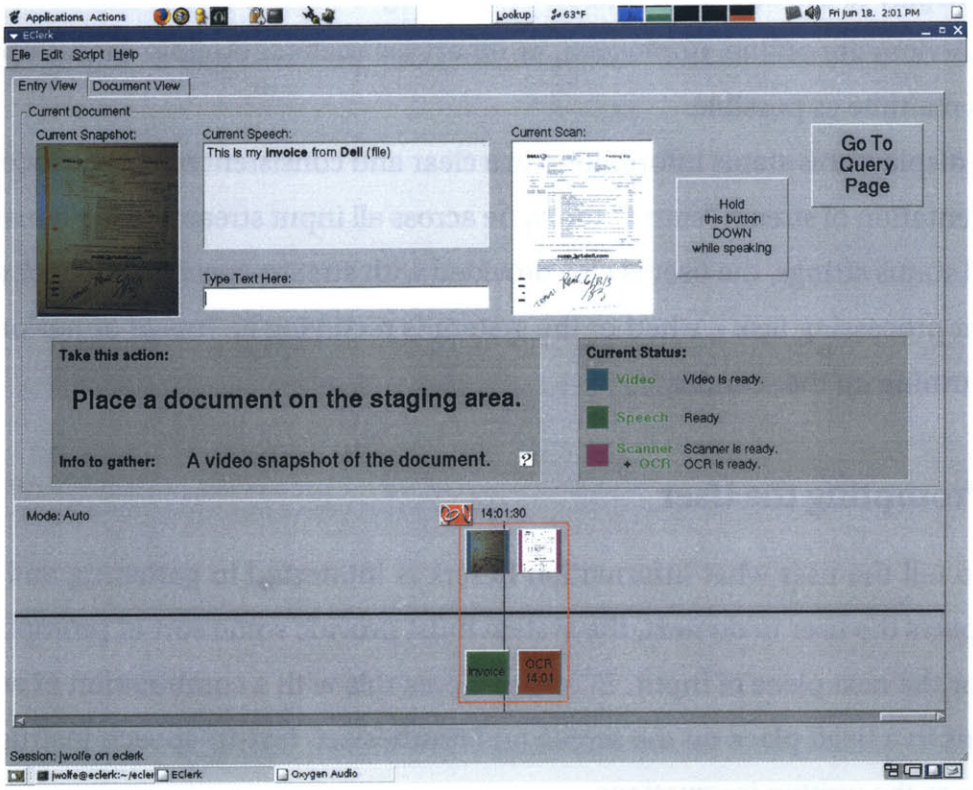

Figure 6-2: EClerk User Interface: Complete Binding. Video snapshot, user utterance, scan, and OCR text from the scan are displayed on the timeline. The resulting virtual document is indicated on the timeline by a border around the input elements making up that document item.

More importantly, users can zoom in on this binding by clicking on any of the thumbnail buttons for items that make up this binding, or "virtual" document (see Figure 6-3). Virtual documents are displayed to the user as a collection with many different views, one for each of the items that makes up the collection. The user can select whichever view is most useful. EClerk picks the default view for any collection by showing images before text and by choosing higher resolution images before lower resolution images of the document.

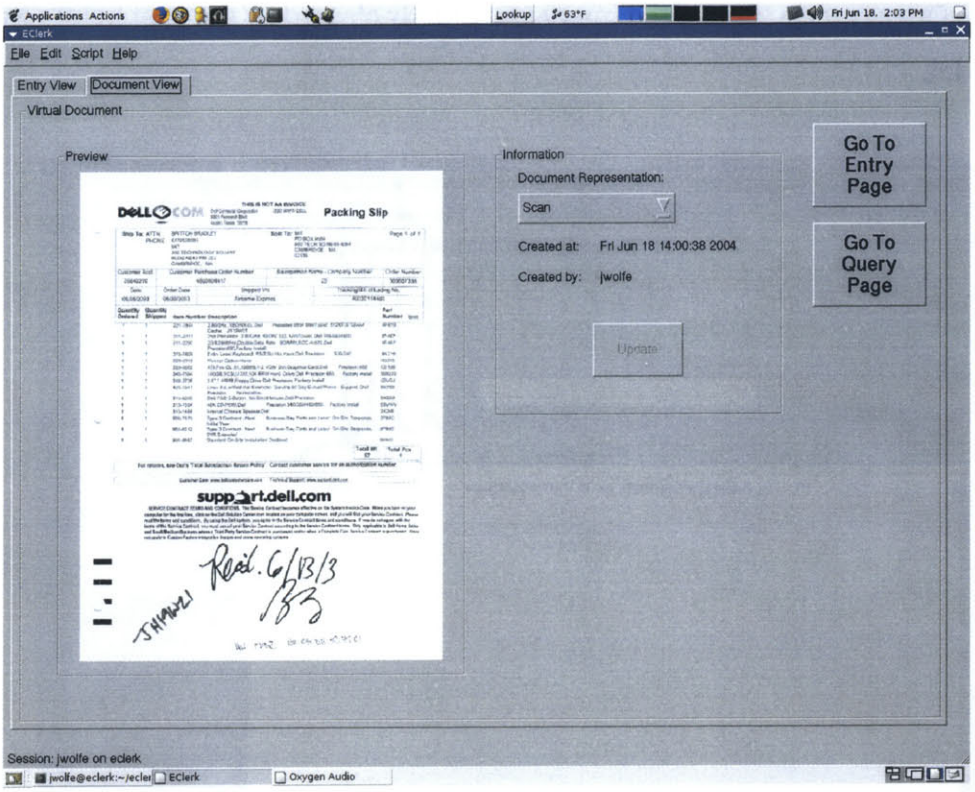

Figure 6-3: EClerk User Interface: Virtual Document View. The user can choose between any document representation available – video snapshot, scan, utterance text, etc. – to view a document preview on the left.

6.2 Query Mode

Query mode allows users to issue queries about all input to the system. In addition, users can restrict search results by entering keywords to match against results. Query results consist of a sorted list of virtual document thumbnails with some additional text information identifying the document. By providing this extra information beyond the thumbnails (i.e. when the document was entered into the system and by which user), the user can more easily disambiguate query results.

Users can easily toggle back and forth between Entry and Query modes. A consistently placed toggle button exists in both Entry and Query modes to take the user seamlessly back and forth between these modes. The user sees the last query issued when

re-entering Query mode (unless the query is explicitly cleared by the user before leaving Query mode).

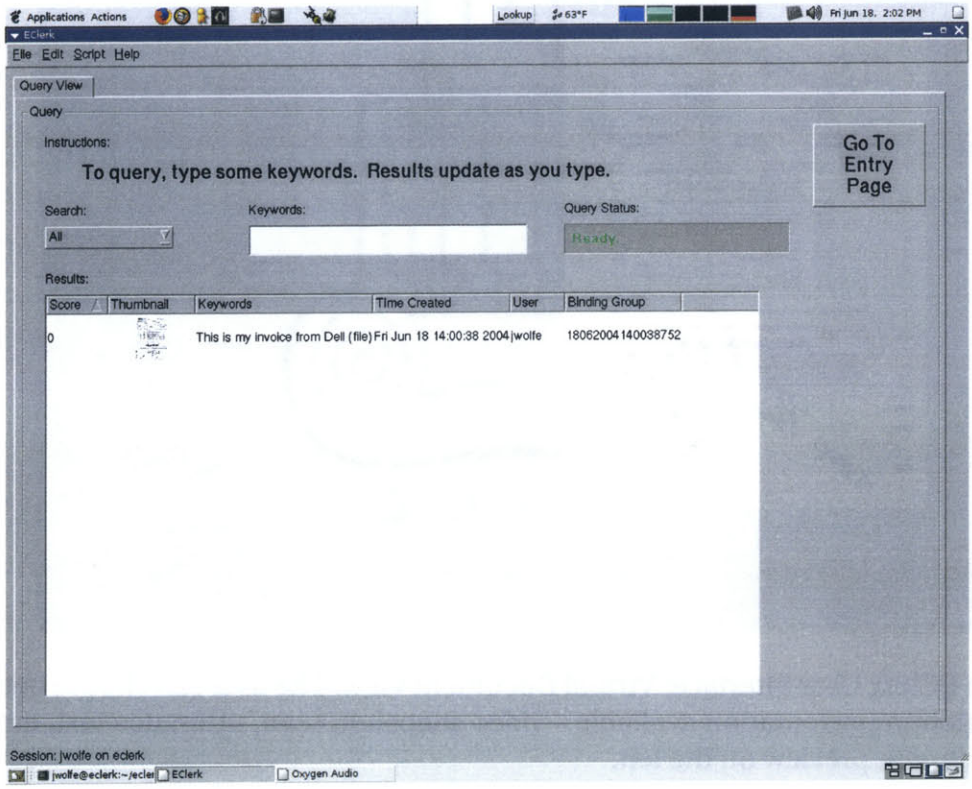

Figure 6-4: EClerk User Interface: Query Mode. Query results are sorted by relevancy and update automatically when the query is updated at the top.

6.2.1 Document Scoring

EClerk uses a simplistic scoring algorithm for sorting the query results. Each document text corpus is matched against the keywords entered by the user, including both the speech text and the text from the character recognition on the high resolution image. Simple word frequency scores the documents.

6.3 Diagnostic Mode

Diagnostic Mode can be enabled using a command line argument in order to provide debugging tools in the user interface. This mode provides a realtime picture of each input stream, extended status for each stream, and status logs. Diagnostic mode also allows advanced users to delete extraneous database records that may build up from debugging new system functionality.

6.4 Data Collection Mode

Data Collection Mode can be enabled using a command line argument in order to provide a stripped down interface simply for collecting data about documents. This mode does not attempt to provide information back to the user so that the user can correct mistakes if they occur – the goal is to immediately give feedback that the document information has been collected in order to collect data on as many documents as possible. This data can be analyzed offline in an effort to gather information about as many new user scenarios and user utterances as possible.

Chapter 7

Results

7.1 Mass Data Collection

EClerk will be deployed in a semi-public manner in Data Collection mode in order to collect as much data as possible about typical user interactions with the system. EClerk will attempt to operate as autonomously as possible in order to make this data collection across multiple users as seamless as possible without administrator interaction. Consequently, the system does not echo every recognized utterance to the user. In addition, EClerk will defer the optical character recognition to offline, where possible. This streamlined process will help to collect as much data as possible.

This data can be analyzed to iterate the speech domain by adding new actions and attributes that users commonly request, in addition to upgrading the user interface of the control software to be more user friendly. This data corpus should be gathered from users with all levels of experience with computers in order get a better cross-section of the eventual user base.

7.2 CSAIL Reimbursements

The first real life situation that EClerk attempts to solve consists of routing business reimbursements to the appropriate parties for resolution. In order to achieve this, EClerk makes use of its generic document processing ability including the ability to send copies of documents to specified individuals. In order to process a reimbursement, the user seeking a reimbursement enters his or her receipt and additional documents into EClerk, and instructs EClerk to send mail to the appropriate person in charge of issuing reimbursements.

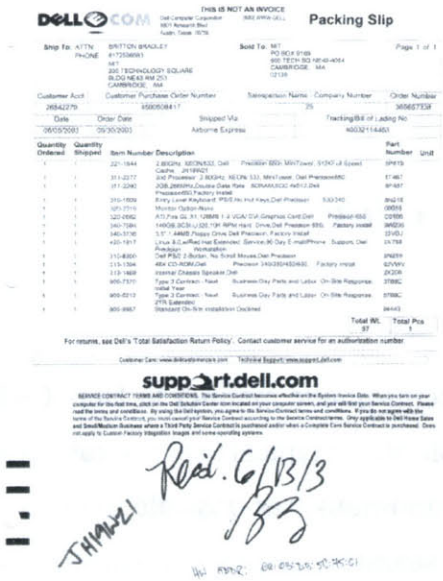

Figure 7-1: Example document submitted as a CSAIL reimbursement – in this case, proof of delivery of a Dell computer.

Because all the documents needed to issue a reimbursement are in electronic form once entered into EClerk, the process is no longer dependent on keeping large amounts of paper around in order to complete a reimbursement. Paper copies of invoices can still be used to sanity check the system and to provide verification for the actual reimbursement amounts, but these copies need not be used or even kept.

With the help of Bryt Bradley, the MIT CSAIL Computer Graphics Group administrative assistant, I hope to use EClerk for submitting reimbursement requests from students in the lab area surrounding EClerk starting in Fall 2004.

7.3 Evolution of the EClerk Workflow

The workflow for using the EClerk system to effectively enter and retrieve documents evolved dramatically from the initial implementation of the user interface. Most important are the improvements stemming from user feedback throughout the evolution process. At various points throughout the development of the system, test users were asked to enter a few documents and interact with the system without any training. This feedback led to many of the interface elements present in the system as described in this document.

First and foremost is the ability to switch seamlessly between EClerk's two modes of operation. Earlier versions of the user interface used different tabs for the different modes. This required users to click on a small tab at the top of the screen in order to switch between modes. In addition, other EClerk modes included the virtual document view and a view for previewing individual input elements. Even though EClerk switched users between modes when an appropriate request was issued, test users were crippled by the tabbed interface because of the small size of the tabs themselves and the number of different modes.

In order to address this navigation problem, EClerk evolved into just two modes, one for document entry and another for querying. Document viewing combined into a single interface that displays previews for all items inside a virtual document instead of separating out a view for individual items. This viewing mechanism is accessible from both Entry and Query modes as necessary. Navigating between the two modes no longer depends on any tabbed interface – large buttons placed in a uniform position on the screen allow for switching between modes and for returning to either mode

from document viewing. This navigation system simplifies getting around EClerk and clarifies to the user what is expected by defining a true "mode" of the system.

As important is EClerk's ability to back off with continuous user feedback where appropriate. The feedback in question consisted mostly of the text-to-speech prompting to the user for what to do next. The scenarios gathered for when this feedback became less helpful and more annoying to the user provided means to adjust both the timing and the feedback text for this prompting. Users who learned the timing of the system more quickly than other users wanted this prompting to back off sooner. Users who still needed the prompting in order to properly complete the workflow did not request that the prompting back off as they depended on it for what to do next. The resulting mechanism – backing off after a certain number of user interruptions – provides the "right" level of feedback for both types of users.

7.4 Adoption of EClerk

In order for EClerk to be adopted as a tool in the office environment, it must be as easy to interact with as speaking directly to a human being. In the case of reimbursements, EClerk must correctly identify the document entered as a document for which a reimbursement request should be made. EClerk uses the user's utterance as a sanity check – it will most likely contain a clue that a reimbursement is the desired action, if not contain the word "reimbursement". Finally, EClerk must present the virtual document to the human on the "other end" in a clear and conspicuous manner in order to ensure the same level of clarity and attention as requests issued in person. The only way to achieve this level of performance is to continue to use the system in a real environment and iterate the workflow and technology that make EClerk as responsive and accurate as possible.

Chapter 8

Future Work

8.1 Invoking Intelligent Agents for Common Tasks

Common tasks like sending email or handling a reimbursement should be easily recognizable to EClerk using similar speech queries and document type analysis. For example, if a document is entered as type "receipt", there is high probability that the user will want to send that document to the person in charge of issuing reimbursements.

Agents developed for each of these common tasks can make the user's life easier by looking for these patters and suggesting actions that seem appropriate. An agent for reimbursements might generalize that documents of type "receipt" are often sent to the same person via email immediately after being entered into the system. Such an agent could then suggest to the user the action of sending any new document of type "receipt" to that person.

Such agents would assist in more than just a business setting. Any users not familiar or comfortable with a full computer interface could use this system to turn paper into electronic form. Family members that want to share any sort of paper documents could use this system (coupled with some sort of encrypted mode of communication like secure email) as a sort of glorified fax machine with the benefit of a simple interface, no dependence on the telephone system, and the ability to make use of an *elec-*

tronic version of the data contained in the document without the hassle of any manual reproduction.

Another possible scenario for an intelligent agent to succeed at anticipating a user's next request involves taking information found on a document and placing in appropriate places to achieve a goal. For example, if a printed document entered into the system contained a tracking number or flight number, an intelligent agent could use some knowledge about what web sites to visit in an attempt to display status information to the user (in much the same way as Amazon displays tracking information to users by retrieving that information and plugging it into a user-recognizable webpage).

8.2 Dynamic Updates to Speech Domain

A more ideal speech recognition system would allow users to dynamically enter information about any unrecognized words and have those domain changes propagate immediately into further recognition. For example, if the system did not know about a new document type (i.e. a "certificate") and did not recognize the user's speech correctly, it would prompt the user to enter in the correct spelling of the particular word spoken and attempt to categorize the word into the domain (with the user's help if necessary).

Not only would this be useful for new document types, it would be especially useful for new attributes like people that EClerk is aware of. In order for EClerk to properly understand a user if a request is made to send a document to a particular person, the system must have that person's name in the domain to be recognized. A more ideal system would immediately take the information from the user on the new person and make that information available for all further speech recognition, in addition to fulfilling the task at hand – sending the document to the new person.

8.3 Learning from Users

EClerk should learn from the usage patterns of users. By associating common document types with actions that are typically requested for such documents, EClerk can learn to anticipate users' needs. In addition, EClerk can use image data collected in conjunction with OCR text to attempt to classify documents without an explicit utterance from the user about the type of document. This should be possible by examining placement and quantity of numbers, and by generalizing on larger, more important words on the document.

8.4 Providing More Accurate Relevance for Queries

EClerk's primary goal is not to be a query engine. Ideally, Haystack (or other information retrieval clients built to effectively query semi-structured data) should be used to successfully issue queries on the document store. EClerk's Query mode operates in a simplistic manner with this document store. Providing a more accurate picture of how relevant each document in the store is to a user query would enhance the user's experience.

Chapter 9

Contributions and Conclusion

9.1 Infrastructure and Integration

EClerk integrates speech technology, information retrieval technology, and an extremely usable interface in order to provide the infrastructure for future work on such an intelligent electronic clerk. This infrastructure is highly modular – the speech computation and the underlying database are not required to reside on the same machine. In addition, the system provides robustness in design, communication, and interface. The modular design of the system allows for modules to be easily added for new input streams or for new functionality. All communication protocols between these components are stateless, providing extra robustness against stale data and system crashes. The user interface provides continuous feedback in order to inform the user about decisions made and to allow the user to correct the system when it makes mistakes.

9.2 Providing a Truly Usable System

EClerk as a proof-of-concept system shows that such a usable system can, in fact, be built. The system's interaction centers around continuous feedback to the user in order to respond very much like an actual human clerk. As speech technologies advance, they

will help to remove more restrictive modes of interaction with EClerk in order to make that interaction as natural as possible. While restricting the domain of recognition to paper documents, it is hoped that the recognition engine can maintain a high level of understanding while continuing to segment speech into actions and attributes, allowing the system to extract important metadata from user utterances. Once deployed, EClerk will remain usable by learning from data collected from users and dynamically adjusting to users' requests. In this way, EClerk will succeed as a truly usable system.

Appendix A

Project Build Instructions

This chapter describes how to checkout, build, and execute the applications discussed in this thesis, including Haystack, Galaxy, and EClerk applications. These instructions assume an account with the MIT Computer Graphics Group, and membership in the *graphics* group. These instructions also assume a Linux command-line environment, running under the bash shell.

A.1 Checkout and Build Instructions

First, check out the CVS source tree of the `eclerk` project. Setup CVS:

```
% export CVSROOT=/c3/eclerk
```

If the desired machine does not have the `/c3` directory mounted as an NFS share, use the following CVSROOT (assuming the username is `jwolfe`) and verify that CVS will use `ssh`:

```
% export CVSROOT=:ext:jwolfe@citydev.csail.mit.edu:/c3/eclerk
% export CVS_RSH=ssh
```

Make sure that this environment variable is set by checking the environment, do this with:

```
% env | grep CVSROOT
```

and verify that the CVSROOT environment variable is properly set. Next, move to the directory where the `eclerk` source tree will be hosted. To checkout to the $ECLERK_DIR directory, type:

```
% cd $ECLERK_DIR
% cvs checkout -P eclerk
```

A.2 Haystack Setup

The current EClerk system uses a current CVS checkout of the Haystack client. This can be obtained from the Information Theory group at MIT's CSAIL using the CVS commands below, or from `http://haystack.csail.mit.edu/`. Verify that your installation contains and properly runs the `ant` target `experiment`. Before starting the EClerk application, invoke this target in a separate terminal window.
Check out Haystack like this:

```
% cd $HAYSTACK_DIR
% cvs -d :ext:eclerk@harrier.csail.mit.edu:/h/haystack/cvsroot checkout
  haystack
```

A.3 Galaxy Setup

The current EClerk system also uses Galaxy version 3.7. This version of Galaxy can be acquired from the Spoken Language Systems group at MIT's CSAIL. Check out Galaxy like this:

```
% cd $GALAXY_DIR
% cvs -d :ext:jwolfe@speech.csail.mit.edu:/papa0/home/sls/src checkout
  distribution
```

Create a symlink to this version's directory:

```
% cd /home/sls/Galaxy
% ln -s Galaxy-v3-7 current
```

In the `speechbuilder` directory under the root directory of the checked out EClerk module there is a file containing the necessary modifications to Galaxy. Move the file `galeclerk.tar` to the `oxygen` directory in the Galaxy source tree. This path should be similar to:

```
/home/sls/Galaxy/current/oxygen
```

Untar the file and remake the source tree. Double check that the file `eclerk.jar` exists in the proper directory:

```
% cd current/oxygen
% cp $ECLERK_DIR/eclerk/speechbuilder/galeclerk.tar ./
% tar -xvf galeclerk.tar
% ls -l /home/sls/Galaxy/current/jar/eclerk.jar
```

The Speech Builder site is at `http://speech.csail.mit.edu/SpeechBuilder`. Download the domain file from the SpeechBuilder website and place it in:

```
/home/sls/Galaxy/SpeechBuilder/users/jwolfe/
```

Untar the file and verify that the symlinks contained within it are proper file system links. The following commands assume that the domain file was downloaded to the $ECLERK_DIR directory and that the username is `jwolfe`:

```
% cd /home/sls/Galaxy/SpeechBuilder/users/jwolfe/
% mv $ECLERK_DIR/jwolfe.tar.gz ./
% tar -xvzf jwolfe.tar.gz
% ls -l DOMAIN.eclerk
```

To start Galaxy:

```
% cd DOMAIN.eclerk
% ./oxclass.cmd 1
```

To start the Java program that allows EClerk to connect to Galaxy's frame relay module, do the following in a separate terminal:

```
% cd /home/sls/Galaxy/current/jar
% java -jar eclerk.jar localhost 60001 eclerk
```

A.4 Linux Kernel Support

If the machine running the EClerk application does not already have support for USB and Firewire devices, the kernel must be rebuilt with the appropriate support. Consult the Kernel HOW-TO for excruciating detail on how to rebuild the kernel for your system. If the system supports multiple processors, include SMP support when the kernel is rebuilt.

A.5 Invoking Applications

Verify that the machine that will compile the EClerk application has the following software (with appropriate versions) installed:

- `libraw1394` version 0.9.0 (or above)
- `libdc1394` version 0.9.4 (or above)
- `Qt` version 3.2.3 (compiled with multithread support)
- `OpenCV` version 0.9.5 (or above)
- `sane-backends` version 1.0.12 (or above)

To check that the versions of Qt and SANE are appropriate, verify the version numbers like this:

```
% moc -v
% scanimage --version
```

To check that the Firewire camera is working properly, run a video capture application like `coriander` (using the `video1394` module for capture) to verify that video capture is successful.
To actually compile and invoke the EClerk application from the root directory of the checked out EClerk module, first execute Qt's meta-compiler in order to create the correct Makefiles for your setup:

```
% qmake
```

Now, make EClerk and run it:

```
% make
% ./eclerk
```

Appendix B

EClerk System Demo

B.1 Video Demonstration

The EClerk video demonstration is available as a separate module under CVS. Verify that the CVSROOT environment variable is properly set, as described in Appendix A. Next, move to the directory where the video will be stored. Note that the video is rather large. To checkout to the $ECLERK_DIR directory, type:

```
% cd $ECLERK_DIR
% cvs checkout -P eclerk-demo
```

In addition to the video, this module contains an archive of the demonstration script used to perform this demo. Copy this file over to the main EClerk directory, unarchive it, and execute the script by opening the `demo.esc` file from the Script menu in EClerk. The commands below assume that both modules were checked out to the $ECLERK_DIR directory:

```
% cd $ECLERK_DIR
% cp eclerk-demo/demo.tar eclerk/demo.tar
% cd eclerk
% tar -xvf demo.tar
% ./eclerk
```

B.2 Storyboard

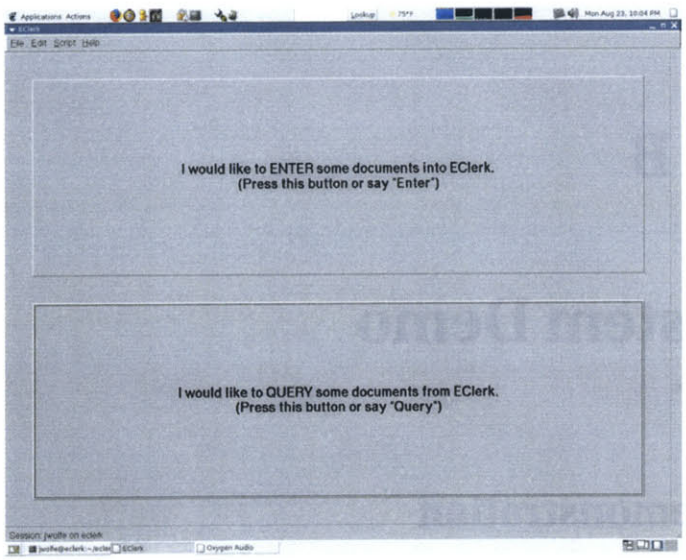

Figure B-1: EClerk is waiting for some interaction with a user.

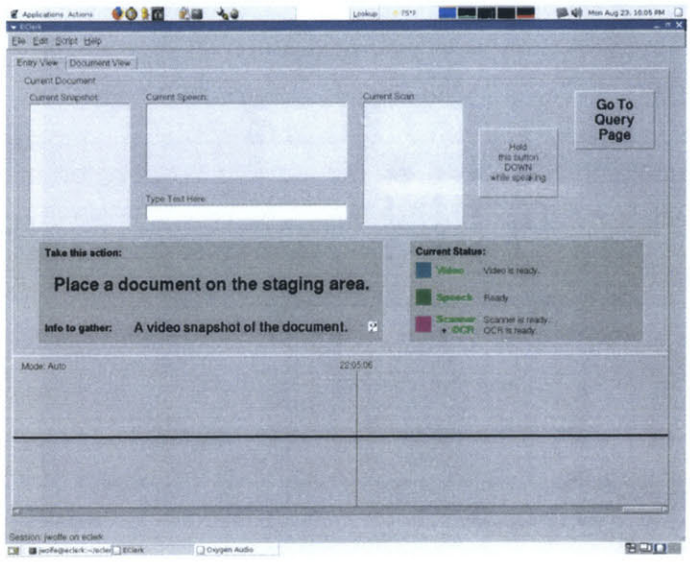

Figure B-2: EClerk Entry Mode is selected. Each input stream is ready for input as indicated by the green status on each stream display.

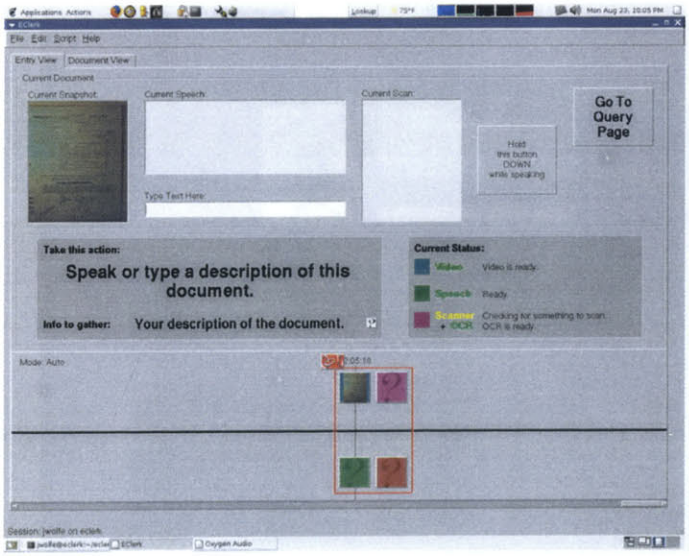

Figure B-3: The first document is entered. EClerk takes a snapshot and asks the user to describe the document. Because EClerk expects other input elements to be paired with this snapshot, a partial virtual document is displayed on the timeline with color-coded input elements waiting to be filled in.

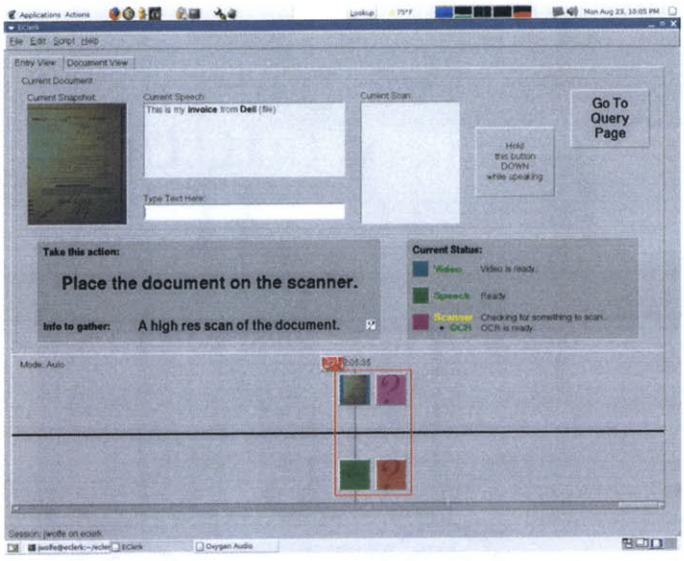

Figure B-4: EClerk uses the Galaxy speech recognizer to interpret the user utterance. The speech text is displayed at the top with key attributes identified in bold – in this case, the document type and a company name. Then EClerk prompts the user to place the document on the scanner.

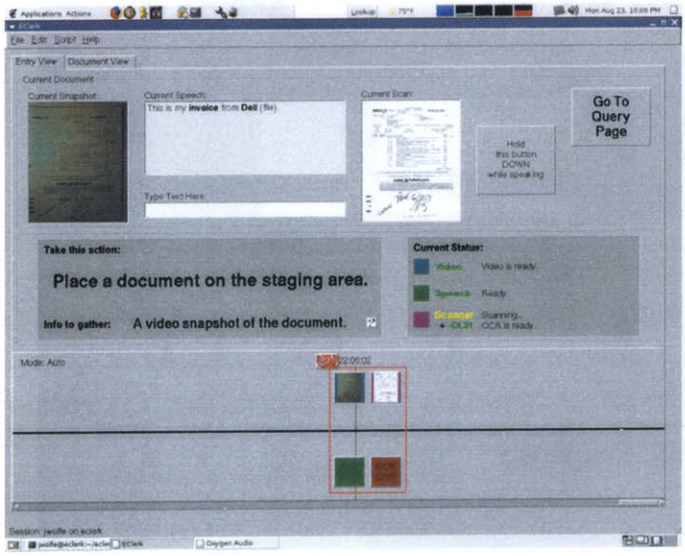

Figure B-5: EClerk completes the entry of the first document – EClerk scans the document and batches the OCR performed on the scan in order to be ready for the next document. EClerk stores all the relationship information about which elements belong to which virtual document in the underlying RDF store.

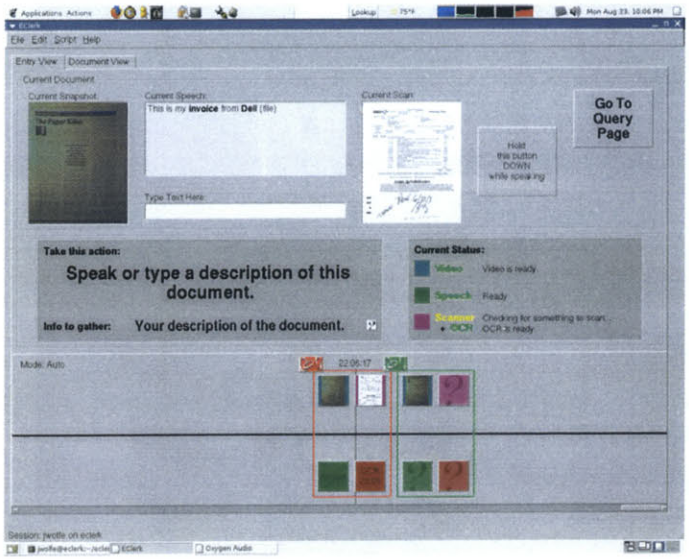

Figure B-6: The next document, an article, is placed on the staging area. A snapshot is taken, and the user is prompted to describe the document. A new virtual document is created on the timeline for this new document.

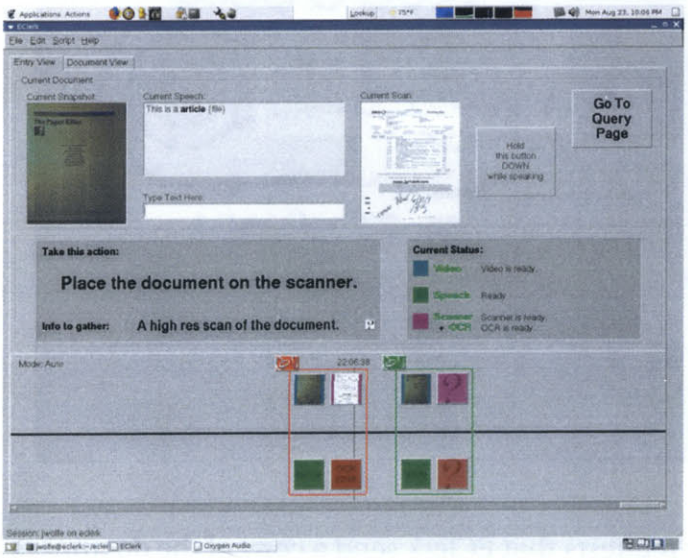

Figure B-7: Once the document is described, the user is prompted to place the document on the scanner, just like the previous document.

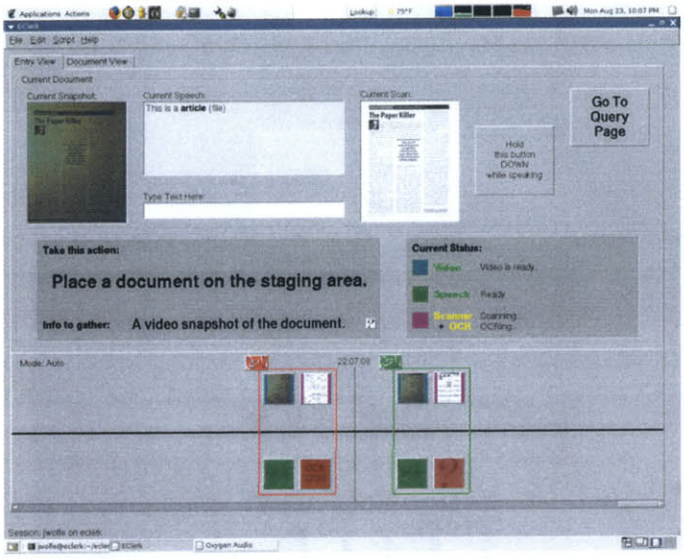

Figure B-8: The second document is scanned, with the OCR batched. Again, the completed document is written to the RDF store by indicating which input elements combine to form the virtual document.

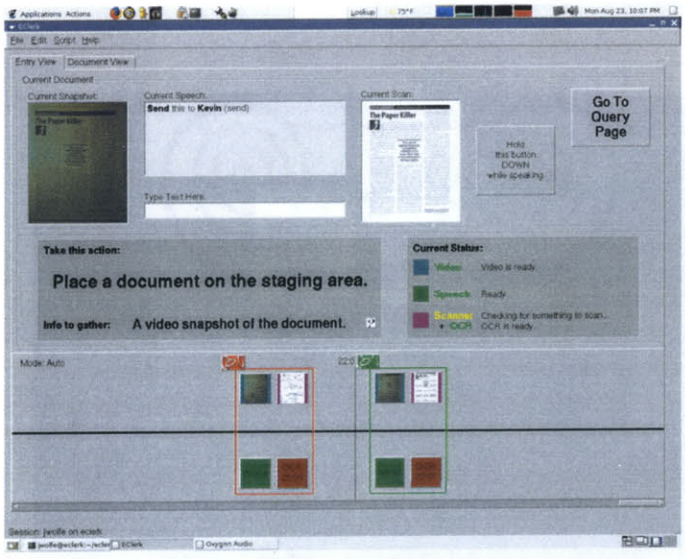

Figure B-9: The user requests that EClerk send a copy of the document to Kevin. EClerk sends an email with a representation of the document attached. Note that EClerk does not treat this utterance as a text entry for the current document because it is a request for action, not true input. EClerk also completes the batched OCR job on the second scan in the background.

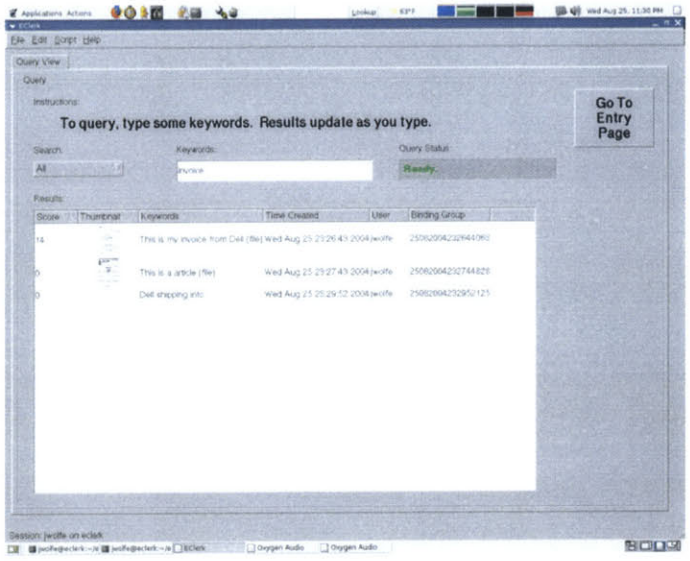

Figure B-10: The user issues a speech query requesting that EClerk look up invoices. EClerk automatically navigates to Query mode and executes the query. Results are sorted in order of relevancy.

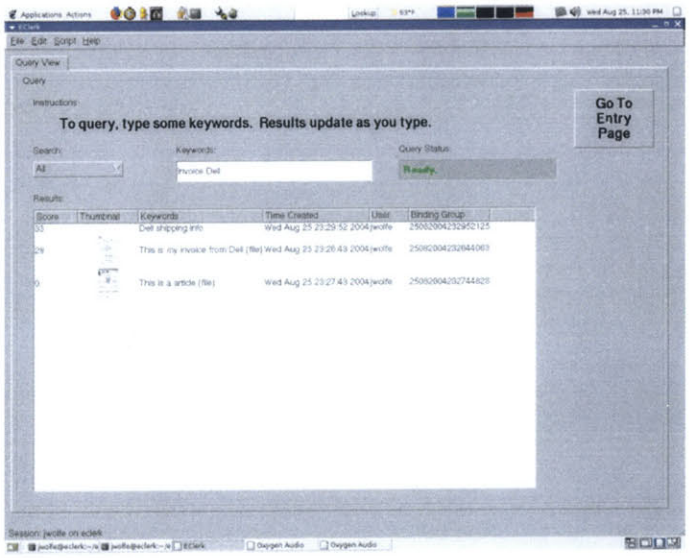

Figure B-11: The user can modify the query by typing into the keyword box at the top. Adding the word "Dell" searches for invoices from Dell. The query results automatically update below.

Bibliography

[1] Bush, Vannevar, "As We May Think" *Atlantic Monthly* 176: 101-108.

[2] Bush, Vannevar, "Memex II" *From Memex to Hypertext: Vannevar Bush and the Mind's Machine*: 165-184. Reprinted from the last menuscript draft, dated August 27, 1959.

[3] Creary, L. and M. VanHilst, "'Where Are the Christmas Decorations?': A Memory Assistant for Storage Locations" *HPL-2001-145, HP Laboratories (Palo Alto, 2001).* `http://www.hpl.hp.com/techreports/2001/HPL-2001-145.pdf`; accessed August 11, 2003.

[4] "Dublin Core Metadata Element Set, Version 1.1: Reference Description" `http://dublincore.org/documents/dces/`

[5] Garfinkel, S., "The Paper Killer" *The Technology Review*, May 2004.

[6] "Haystack Home" `http://haystack.csail.mit.edu/`

[7] Huynh, D., D. Karger, and D. Quan, "Haystack: A Platform for Creating, Organizing and Visualizing Information Using RDF." *Semantic Web Workshop, The Eleventh World Wide Web Conference 2002.* `http://haystack.csail.mit.edu/papers/sww02.pdf`; accessed August 10, 2003.

[8] "MIT Spoken Language Systems Group: DARPA Sponsored Research" `http://www.sls.csail.mit.edu/sls/sponsorship/DARPAactivities.html`

[9] Pogue, D. "Scanning a Paperless Horizon" *New York Times* - State of the Art, July 15, 2004.

[10] Polifroni, J. and S. Seneff, "GALAXY-II as an Architecture for Spoken Dialogue Evaluation." *Proc. LREC '00, (Athens, Green, 2000).* `http://www.sls.csail.mit.edu/sls/publications/2000/lrec-2000.pdf`; accessed August 10, 2003.

[11] "Qt Reference Documentation" `http://doc.trolltech.com/3.2/`

[12] Quan, D. "Metadata Programming in Adenine" `http://haystack.csail.mit.edu/documentation/adenine.pdf`

[13] Quan, D., D. Huynh, D. Karger, "Haystack: A Platform for Authoring End User Semantic Web Applications." `http://haystack.csail.mit.edu/papers/www2003-developer.pdf`; accessed August 10, 2003.

[14] Rhodes, J.R., T. Starner, "Remembrance Agent: A Continuously Running Automated Information Retrieval System." *Proceedings of the First International Conference on The Practical Application of Intelligent Agents and Multi Agent Technology (PAAM '96)* London, UK, April 1996, pp.487-495.

[15] Seneff, S., E. Hurley, R. Lau, C. Pao, P. Schmid, and V. Zue, "GALAXY-II: A Reference Architecture for Conversational System Development." *Proc. ICSLP '98:931-934, (Sydney, Australia, 1998).* `http://www.sls.csail.mit.edu/sls/publications/1998/icslp98-galaxy.pdf`; accessed August 10, 2003.

[16] Seneff, S., R. Lau, J. Polifroni, "Organization, Communication, and Control in the GALAXY-II Converational System." *Proc. Eurospeech '99 (Budapest, Hungary, Sept 1999).* `http://www.sls.csail.mit.edu/sls/publications/1999/eurospeech99-seneff.pdf`; accessed August 10, 2003.

[17] Teller, S., D. Karger, J. Glass, "Smart File Cabinet Prototype." *(Cambridge, Massachusetts, Dec 2002).* `http://graphics.csail.mit.edu/sfc/smartcabinet.pdf`; accessed August 10, 2003.

[18] Weinstein, E., "SpeechBuilder: Facilitating Spoken Dialogue System Development" *M.Eng. thesis, MIT Department of Electrical Engineering and Computer Science (May 2001).* http://www.sls.csail.mit.edu/sls/publications/2001/EugeneWeinsteinMEngThesis.pdf; accessed August 11, 2003.

[19] "Xerox Scanners and Projectors" http://www.xeroxscanners.com/